Master of Science Thesis

Simplifying Dynamic Fault Trees by Graph Rewriting

Sebastian Junges

March 25, 2015

Chair for Software Modeling and Verification
RWTH Aachen University

Supervisors:
Prof. Dr. Ir. Joost-Pieter Katoen
Dr. Mariëlle Stoelinga (University of Twente)

Abstract

The thesis examines rewriting of *Dynamic Fault Trees* (DFTs) to accelerate their quantitative evaluation. Fault trees are a prominent model in the context of *reliability engineering*, and have been widely adopted by industry. *Dynamic* fault trees extend the expressive power of regular fault trees to allow faithfully modelling common patterns as shared spare components. A major drawback of the model is that it is subject to the state space explosion problem. In the thesis, minimising the DFT is presented in order to alleviate this issue. In particular, the objective is to create a formal framework that allows rewriting DFTs by replacing predefined patterns in the DFTs with other patterns — that is, by the application of *graph rewriting*.

One cornerstone for this framework is a *semantics for DFTs* that allows intuitive yet efficient reasoning about DFTs. In a comprehensive survey, the complexity of DFTs is uncovered and illustrated by a range of semantic intricacies. Supported by the lessons learned from the survey, a denotational style semantics is given and used to characterise DFTs and define equivalence classes upon it.

The *rewrite framework* is based upon a reduction to standard graph rewriting. This allows the rewrite rules to be defined in terms of DFT patterns, while exploiting the well-researched field of graph rewriting — including the available tool-support. Rules in the framework can be proven correct by utilising one of the presented theorems which show that simple criteria on the rules imply the equivalence of input and output of the rewrite procedure. The wide range of rules definable in the framework is illustrated by a selection of rewrite rules, which form a minimal basis.

The last part of the thesis demonstrates the *practical relevance* of the framework. A prototype of a fully-automatised tool chain for rewriting is applied on a broad range of benchmarks, which are based on case studies from literature. We compare the quantitative analysis of the original DFTs with their rewritten counterparts. Many simplified instances are analysed 10 times faster, some even up to a factor 100. The memory consumption is drastically reduced. As a consequence, DFTs up to four times larger than feasible without simplification can be analysed within the same time and memory limits.

Preface

While writing the last sentences of my thesis, I'm both happy and proud that I finished my thesis about the simplification of dynamic fault trees. A topic which, before I started it, would never have been my first choice. However, in retrospective, I'm very glad that I was allowed to work on it. During my visit at the University of Twente, I started working on some properties of the underlying models for dynamic fault trees. Then, Dennis Guck introduced a particular problem: the generation of the underlying models was a bottleneck in the analysis of fault trees. Together with Mariëlle Stoelinga and Arend Rensink the aimed to alleviate this problem by graph rewriting. They invited me to join their discussion...

Acknowledgements First of all, I would like to thank my supervisors, first of all for giving me the opportunity to visit Twente. Joost-Pieter Katoen, thanks for the freedom you gave me and the patience to let me work on it until it was done, as well as the many valuable hints and the detailed feedback. Mariëlle Stoelinga, thanks for the long and fruitful discussions and all the feedback you gave me, also after I left Twente again. Furthermore, I would like to thank Arend Rensink for all his help, on Groove and on several other thesis-related questions, and Dennis Guck, for the extraordinary long list of help I received from you on an almost daily basis. Enno Ruijters, I really appreciated how you were always there to discuss absurd fault trees and questions about them. Before I started on dynamic fault trees, I had great discussions with Nils Jansen and Christian Dehnert, for which I'm grateful.

I would like to thank the people from the MOVES group in Aachen for all their support, and the FMT group for making me feel very welcome at Twente. Furthermore, I would like to thank Erika Ábrahám and the whole THS group for introducing me to science and the opportunities they gave me — it was a huge kick start and helped me write this thesis.

Ultimately, besides writing a thesis, life goes on. I'm very glad I've such supportive and helpful friends. The way my parents support and motivate me is extraordinary, and I really appreciate that. Last but not least, Irma, thank you for your wonderful support!

*

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Sebastian Junges
Aachen, den 24. 3. 2015

Contents

1. Introduction

This chapter introduces fault tree analysis and dynamic fault trees as a powerful model used within fault tree analysis. The first section motivates the use of fault trees and dynamic fault trees in particular. Three hypotheses expound the benefits of simplifying fault trees and the charm of using graph rewriting to achieve this. The second section gives an overview of the derived objectives for the thesis. The third section provides a brief overview of the related literature and the last section outlines the further contents of the thesis.

1.1. Motivation

Individuals and companies - even society as a whole - depends ever more on increasingly complex systems. So-called safety critical systems, for which reliable operation is key to prevent life-threatening situations are manifold. Besides typical examples as (nuclear) power plants and avionics, also the power grid and phone lines are safety critical systems. Other than safety-issues, unreliability of systems may cause tremendous financial loss and seriously harm the reputation of the trade mark (cf. the Ford Pinto fuel tank and the Pentium floating-point unit bug). To ensure that systems are reliable, standards and certificates have been introduced. For these certificates, it is important that the reliability of system can be assessed. A simple approach to improve the reliability of a system is the introduction of redundancy. For functionality in a system that is crucial for correct operation, multiple components are installed. Only a subset of these components are required to be operational to keep the system as a whole operational. A well-known example are the multiple sensors and actuators in aircraft. The spare wheel commonly found in the trunk of a car is also an example of redundancy, as of the five wheels present, only four are required for the car to be operational. Redundancy, however, is a costly measure. Assessing the reliability allows for better informed decisions when it comes to adding redundancy.

The assessment of the reliability of — especially safety-critical — systems has been a field of research since the 1920s and has evolved ever since. Two dedicated ways to describe and analyse the reliability of a complex system have emerged [RH04], *Failure Modes and Effects Analysis* (FMEA) and *Fault Tree Analysis* (FTA). Whereas FMEA [Sta03] works *bottum-up*, that is, it considers the effect of a component failure on the system, FTA (cf. [VS02]) considers a failure of the system and which components contribute to this failure, i.e. following a *top-down* approach. Notice that therefore, only FTA is suitable to describe system failures that are due to a combination of components failing.

FTA was introduced in the 1960s at Bell Labs during the development of rocket launch control systems [Eri99], and subsequently adopted in avionics and later in the nuclear reactor industry. It was put in the limelight in the reports of some famous accidents, among them the Apollo 1 launch pad fire (1967) [Apo67], the Three Mile Island partial meltdown (1979) [The79], the space shuttle disasters Challenger (1986)[The86] and Colombia (2003) [Col03] and the Trans World Airlines Flight 800 in-flight breakup (1996) [Nat00]. Its use has been enforced by several authorities, e.g. the US Federal Avionics Administration (FAA), the US Nuclear Regulatory Commision (NRC) and the US National Aeronautics and Space Administration (NASA). It has been standardised by the International Electrotechnical Commision (IEC) [IEC60050-191] and its use throughout industry is widespread. Outside of avionics and nuclear power plants, FTA is known to be used in the automotive industry [Lam04; Sch09], miner safety [Goo88; GMKA14], power system dependability [VCM09] and in railway engineering [CHM07; GKSL+14].

FTA is a methodology consisting of several steps. The three most relevant steps are listed here. In the first step a system failure is described. In the second step, a model is constructed which reflects the relation of this system failure with the failures of specific components. This model is called a fault tree. In the third step, the fault tree is evaluated to assess several properties of the system failure. Such properties can be qualitative, e.g. describing the minimum number of leaf elements that are required to fail before the system fails, or quantitative, e.g. the reliability of the system given the failure distributions for the leaf nodes. Reliability is the probability that the system does

not exhibit the failure under inspection up to a given time horizon. In this thesis, such quantitative (more precisely, stochastic) properties are investigated.

Fault trees are a *graphical* specification language which *hierarchically* define the system failure in terms of subsystem failures. Subsystems which are not further divided are called *components*. In a fault tree, the top-most level corresponds to the system failure and the bottom-most level corresponds to the failures of a component. Failures are then propagated bottom-up. Each inner node of the tree (called a *gate*) represents that the failure of its (sub)subsystems cause the (sub)systems corresponding to the node to fail. At which point the failure is propagated depends on the type of the gate. Several types are available, e.g. stating that all subsystems have to fail before the failure is propagated (AND) or that the first subsystem failure is propagated (OR). The construction is started on the top-most level. All current leaves are investigated. Either, the node corresponds to a system, which is split into further subsystems. Then, its failure is expressed by a specific combination (encoded by the used gate) of its children, which are added to the tree. Otherwise, the node corresponds to a components and is not split up further. Such nodes are called *basic events*.

The commonly used fault trees are also called static fault trees (SFTs), where static corresponds to the control-theoretic notion of static systems, that is, the system is history independent. In the context of fault trees this means that the set of failed components at time t uniquely determines whether the system failure occurs at time t, and that the failures of components are independent of each other. This is a severe restriction of the expressive power of static fault trees, as many systems, especially those with redundant components, are not static in nature. For example, consider the spare wheel in the trunk of a car, whose tire is much more likely to get flat after the wheel is put into operation, that is, after the failure of an original wheel. In many cases, such systems can be modelled by static fault trees which then under-approximate the reliability of the system. In the context of the spare wheel, it can be assumed that the likelihood of a flat tire is fixed and independ of the tire being in use or not flat tire is always as likely as it is when the wheel is operational. However, under-approximating the reliability of a system may render the whole analysis useless. In fact, during early Apollo missions, the NASA used FTA and related methods to compute the reliability for a full mission success - bringing people to the moon and back again. The results were so small that NASA abandoned using the FTA until the Challenger disaster [VS02]. To overcome this restriction, *dynamic* fault trees have been introduced in which the history of failures affects whether a system fails. In contrast to static fault trees, the dynamic counterparts are extended with several gates, which, e.g., require a specific ordering of the basic events to happen or only activate specific components after the occurrence of other failures. Thereby, they enable the modeller to account for order-specific behaviour and failure rates depending on the current state. This yields fault trees which potentially model the real system behaviour more faithfully and thereby yield more realistic figures when evaluating the fault tree. Dynamic Fault Trees have been adopted by industry and are used by, among others, Airbus [BCCK+10], BMW [Sch09] and the NASA [VS02].

As often, the less expressive model of static fault trees are much easier to analyse. That is, static fault trees can be put into normal form by basic Boolean manipulations and the reliability can then be calculated using high-school mathematics [VS02]. All this is possible in polynomial time and space. On the other hand, dynamic fault trees possess an internal state space. This state space can be made explicit by transforming a DFT into some kind of transition system. In the context of the probabilistic properties, a Markov chain lends itself as a straightforward model for the underlying transition system. Evaluating properties on the DFT then boils down to first constructing the underlying state space, reformulating the property to refer to the underlying state space and then analysing this property on the underlying state space.

The analysis whether a given transition system satisfies a particular property is called *model checking*. Several references to model checking problems have been made since the late 1950s. The modern version of this research field has been pioneered by Clark and Emerson [CE82], and Quielle and Sifakis [QS82], Clark, Emerson and Sifakis later won the prestigious Turing award for their work regarding model checking. Although several variants exist, model checking such transitions systems generally boils down to extensive search through the state space.

This traditional form of model checking handles qualitative aspects. Inspired by the success story, several extensions for quantitative aspects have been proposed, most notably for systems involving time, e.g. in [AD94], or probability, e.g. in [CY95]. Such methods are thus readily available to evaluate DFTs. Their performance largely depends on the size of the underlying Markov chain.

Reviewing the process to create dynamic fault trees, we observe that industrial DFTs are not

created with a small outcome as objective - instead the focus is on easy-to-review fault trees. The redundancy in the fault tree comes at large computational cost during the evaluation.

Hypothesis 1

Dynamic fault trees, generated mechanically or according the standardised guidelines, can be significantly reduced by the application of rewriting.

The state-of-the-art algorithm for the quantitative analysis of DFTs (implemented in DFTCalc [ABBG+13]) creates the underlying state space using a compositional approach, where each gate is translated into a Markov chain. The underlying Markov chain is then obtained by taking the parallel composition. With the state space explosion, we get a state space exponential in the number of gates. To alleviate this, the algorithm applies abstraction methods such as *bisimulation* after the composition of two chains. While this drastically reduces the state space for the model checking procedure, the actual generation of the state space becomes the bottleneck for the algorithm. The time and memory consumption for the generation, and thus for the overall procedure are directly affected by the size of the state space.

Hypothesis 2

The reduction of the size of the DFT has a major influence on the run time of the quantitative evaluation of the DFT.

Although different in nature, the number of gates in static FTs directly affects the performance of their analysis algorithms. Most known algorithms exploit some Boolean manipulations to minimise the SFT. While Boolean algebra does not suffice to reduce dynamic FTs, other methods might be very well applicable to reduce the DFTs to make them more suitable for evaluation, while their creation can still follow the engineering practice of easy-to-understand and structured fault trees.

Whereas SFTs are regularly formalised as propositional formulas and their Boolean manipulation can be described as term rewriting, DFTs are usually formalised represented as graphs. This naturally leads to the idea of using *graph rewriting* to describe simplification steps of DFTs.

Graph rewriting [AEHH+99] is an active area of research used in several different contexts. The idea behind graph rewriting is simple. Given a graph (a host) and a rule consisting of two patterns (presented as graphs), a left-hand side (lhs) and a right-hand side (rhs), an algorithm looks for the existence of the lhs pattern within the host (matching), and replaces it by the rhs, yielding a new graph. To replace arbitrary directed acyclic graphs, the quite general notion of algebraic rewriting [Ehr79] is a good choice. The simpler single-pushout approach (SPO) has as a downside that dangling edges are removed, thereby actually changing the neighbourhood of nodes in the graph which are not matched. The double-pushout approach for rewriting simplifies the characterisation of the result as a stricter notion of locality is enforced. Tool support for graph rewriting is readily available. Among other features, Groove [GdRZ+12] has support for typed nodes and rapid (graphical) prototyping.

Hypothesis 3

Graph rewriting is a suitable technique for both formalising and implementing the rewriting of DFTs.

1.2. Objective

Based on the motivation given above, a series of objectives have been developed which are answered in this thesis. We briefly present the objectives — given in the boxes below.

As DFTs are a special family of graphs, DFT rewriting can be implemented analogously to graph rewriting. Given a DFT and a rule consisting of two patterns (given as partial DFTs), an algorithm looks for the existence of the lhs pattern and replaces it, yielding a new DFT. In order to exploit the well-researched theory and algorithms for graph-rewriting, it is beneficial to let the algorithm translate both the host DFT and the two partial DFTs to a representation of them in standard graphs, and then apply standard graph rewriting. The resulting graph can be easily translated back to a

DFT afterwards.

Objective 1

Define DFT rewriting by means of graph rewriting.

The result, however, is not necessarily a syntactically correct DFT. Moreover, as we want to use the rewritten DFTs to assess quantitative properties of the original DFT, these properties should be preserved.

Objective 2

Characterise under which circumstances rewriting a DFT yields a syntactically correct. DFT.

Objective 3

Characterise under which circumstances rewriting a DFT preserves the quantitative properties of interest.

To formally prove that specific properties are preserved, a precise meaning of both model and property are required. While a broad range of semantics is available from the literature, these are not necessarily suitable for proving the requirements above.

Objective 4

Define a semantics for DFTs suitable to prove that the properties of interest are preserved.

These semantics should agree with, or closely resemble, the common interpretation of DFTs found in the literature. To prepare the development of a semantics, it is especially interesting to describe the intricacies of these interpretations.

Objective 5

Survey the common interpretation of DFTs and uncover potential caveats.

With these ingredients, case studies from the literature can be simplified. Ultimately, we want to show that the rewriting is indeed an effective preprocessing step.

Objective 6

Examine the practical relevance of the rewriting.

1.3. Related work

Semantics for DFTs are presented by several authors [BC04; BCS07c; BD05b; BD05a; BD06; Cod05; CSD00; MPBV+06; MRLB10; Wal09; RS14]. A complete overview and details of the semantics are given later, in Section 3.5 on page 67. Here, we briefly give some pointers. The use of FTs in reliability engineering is well-explained in the *NASA handbook of Fault Trees* [VS02] and is an excellent starting point on (the use of) DFTs. Of the various semantics, the operational semantics described in [CSD00] and compositional semantics given by Boudali *et al.* [BCS07c] stand out by the concise definitions and large class of DFTs described. Several notions and ideas to define the semantics for DFTs have been adapted from those papers. Furthermore, graph rewriting is used in the context of DFTs by [Cod05], where DFTs are translated to GSPN by means of graph rewriting. Simplification of DFTs is considered by Merle *et al.* [MRLB10], who use an extension of Boolean algebra to rewrite DFTs. The result, however, is not a DFT anymore, but a algebraic expression. Furthermore, they present a simplification rule in [MRL10]. While this rule is very effective, the rule is only applicable on a restricted class of DFTs.

1.4. Outline of the thesis

The remainder of this thesis is structured in six further chapters. In Chapter 2 on page 7 we briefly discuss results from probability theory, theory about Markov automata and graph rewriting, both theoretical and applied. The results presented there are used throughout the thesis. Besides a refresher, its main purpose is to fix some notation. In Chapter 3 on page 27 we discuss fault trees in much more detail. This includes a discussion on the use of fault trees, their intuitive meaning and different approaches on the formalisation of DFTs (Objective 5). We additionally introduce a couple of case studies of real-world systems found in the literature. From this chapter, we conclude that several, incompatible, semantics exist. The chapter provides an overview and discusses the practical relevance of the different semantics. In Chapter 4 on page 71, we give our own formalisation of DFTs (Objective 4). We include a selection of statements about the behaviour of DFTs and their proofs. These statements serve the purpose of showing that the semantics agree with several intuitive properties, the proofs show the straightforward argumentation the semantics allow. In Chapter 5 on page 113, we develop a formal framework which allows to specify rewrite rules on DFTs. We illustrate the framework by a variety of rewrite rules already embedded in this framework (Objective 1-3) The chapter shows that the correct application of rewrite rules is hard in general, as many practical rules are only applicable on restricted set of DFTs. In Chapter 6 on page 161, we show the practical relevance of the rewrite rules by automatically simplifying the formerly introduced case studies with Groove (Objective 6). The experiments, based on case studies from the literature, show major improvements in computational costs, thereby confirming our hypotheses. In Chapter 7 on page 179 we conclude the thesis with a discussion of the contents and an outlook with some ideas for future work.

2. Preliminaries

In this chapter, we discuss the required background for this thesis.

We use elementary set operations and relations on sets, which are both covered in, e.g. [Hal60]. We use the common notation, which is briefly given in e.g. [BK08]. Furthermore, we use basic notations from first-order logic, which is covered in, e.g. [EFT96] and assume some familiarity with automata theory, as covered in, e.g. [HMU06]. Some further notation is introduced here. The remainder of the chapter covers basics of probability theory, Markov automata, and graph rewriting from both a theoretical and a practical point of view.

Notation Given a function $f\colon X \to Y$, we denote the *range of* f, i.e. the set $\{y \in Y \mid \exists x \in X\ f(x) = y\}$, as $\mathrm{Ran}(f)$. Given a partial function $g\colon X \nrightarrow Y$, we denote the *domain of* g, i.e. the set $\{x \in X \mid g(x) \neq \bot\}$, as $\mathrm{Dom}(f)$. Given a function $h\colon X \to (Y \to Z)$, we often write $h(x, y)$ to denote $h(x)(y)$.

We use finite and infinite words over finite alphabets. We use the standard notation for composing words via regular expressions, for both finite and infinite words[1]. We denote the set of all finite and infinite words over an alphabet Σ with $\Sigma^{*/\omega} = \Sigma^* \cup \Sigma^\omega$. Let $w = \sigma_1 \dots \sigma_n$ be a finite word. We define $w_i = \sigma_i$ for $1 \leq i \leq n$. Furthermore, $|w| = n$ and $w_\downarrow = w_n$.

We denote the powerset of a set A with $\mathcal{P}(A)$. All subsets of A of cardinality k are given by $\mathcal{P}_k(A)$.

Special functions The heaviside function $u\colon \mathbb{R} \to \mathbb{R}$ is given by $u(x) = 0$ if $x < 0$ and $u(x) = 1$ otherwise. We use $\mathbf{0}$ to denote a function from the reals to 0, i.e. $\mathbf{0}\colon \mathbb{R} \to \{0\}$.

2.1. Stochastics

A brief introduction into the required probability theory is given by Timmer in [Tim13] and Baier and Katoen in [BK08]. An in-depth discussion of probability theory is given by, e.g. Ross in [Ros10], and Ash and Doléans-Dade in [AD00]. Here, we introduce the for this thesis most relevant results and thereby fix some notation. The presented facts follow [Tim13].

We informally introduce a *random variable* as a function which maps all possible *outcomes* of an experiment to a set of values representing the *observations*. As an example, consider the random variable which maps all dice rolls (all outcomes) to either *odd* or *even*(observations), by mapping odd dice rolls to *odd* and all even dice rolls to *even*.

In this context, we are interested in the probability of an observation, i.e. we are interested in the probability that the random variable X evaluates to a value in a subset E of the possible observations. We write $\Pr(X \in E)$. The notations $\Pr(X = e)$ and $\Pr(X < e)$ for some observation e is used to denote $\Pr(X = \{e\})$ and $\Pr(X = \{e' \in E | e' < e\})$, respectively. Furthermore, the probability that X evaluates to some value in E under the assumption that X evaluates to some value in E' is denoted $\Pr(X \in E \mid X \in E')$.

In this thesis, we describe this by *probability distributions*, which yield a probability for each observation. We focus on two classes of random variables, the discrete random variables, which have a countable range[2], and continuous random variables, which have a continuous range. We call the range of a random variable the *sample space*.

Discrete probability theory In the discrete case, we assume a *discrete random variable* $X : \Omega \to \mathbb{R}$ with $S = \mathrm{Ran}(X)$ countable.

Definition 2.1 (Discrete probability distribution)**.** Given a countable sample space S, a *discrete probability distribution* (*distribution*)) $\mu\colon S \to [0, 1]$ is a function such that $\sum_{s \in S} f(s) = 1$. ■

[1] For more information about these ω-regular languages, we refer to [Tho90].

[2] We follow the definition from Ash in [AD00] here

Any discrete random variable can be described by a discrete probability distribution. If the random variable is described by

$$\Pr(X = s) = \mu(s)$$

we write that X is distributed according to μ. The set of all discrete probability distributions over some countable sample space is denoted $\text{Distr}(S)$.

The set $\text{supp}(\mu) = \{s \in S \mid \mu(s) > 0\}$ is called the *support of* μ. A distribution μ with a singleton set $\{s\} \subseteq S$ as support is called a *Dirac-distribution*, which we denote with $\mathbb{1}_s$. It holds that $\mu(s) = 1$. Given an equivalence relation $R \subseteq S \times S$ on S, we write $\mu \equiv_R \mu'$ if for all $s \in S$, $\sum_{s' \in [s]} \mu(s') = \sum_{s' \in [s]} \mu(s')$.

Continuous probability theory In the continuous case, we assume a *continuous random variable* X. In this context, we assume that X thus has a sample space $\mathbb{R}$ and that a probability is assigned to each interval $[a,b]$. We assume that the probability of point intervals $[a, a]$ is always zero. Therefore, the probability assigned to a closed interval $[a, b]$ equals the probability assigned to the corresponding open interval (a, b). The random variable is then specified by a function called the *probability density function*.

Definition 2.2 (Probability density function)**.** Given an interval $[a, b] \subset \mathbb{R}$ and a random variable X, a function $f(x) : \mathbb{R} \to \mathbb{R}_{\geq 0}$ with $\int_{-\infty}^{\infty} f(x)dx$ and

$$\Pr(X \in [a, b]) = \int_a^b f(x)dx$$

is the *probability density function* (*density*) of X. ■

It is common to describe the probability that the observation is smaller than some given value.

Definition 2.3 (Cumulative distribution function)**.** The *cumulative distribution function* for a random variable X with probability density function f is given by

$$F(x) = \int_{-\infty}^{x} f(y)dy$$ ■

For a random variable X with cumulative distribution function $F(x)$, it holds that

$$F(x) = \Pr(X < x).$$

We furthermore use a weighted average over the sample space of a continuous random variables, which is the *expected value*.

Definition 2.4 (Expected Value)**.** The *expected value* $\mathbb{E}(X)$ of a random variable X with a probability density function $f(x)$ is given by

$$\mathbb{E}(X) = \int_{-\infty}^{\infty} x \cdot f(x)dx.$$ ■

It is important to notice that the expected value does not always exist.

In this thesis, we focus on the *exponential distribution*. This distribution has several interesting properties, which are discussed in-depth by Balakrishnan and Basu in [BB96]. We describe the most relevant facts.

Definition 2.5 (Exponential Distribution)**.** A random variable is distributed according to an *exponential distribution* if it is described by a probability density function $f(x)$ such that

$$f(x) = \lambda e^{-\lambda x} \cdot u(x)$$

for some $\lambda > 0$. ■

The λ is called the *rate* of the exponential distribution. The cumulative distribution function is given by $F(x) = (1 - e^{-\lambda x}) \cdot u(x)$. A random variable which is distributed according to an exponential distribution is an exponentially distributed random variable.

The exponential distribution is the only *memoryless* distribution (cf. [BB96]). If X is a continuous random variable and

$$\Pr(X > x + y \mid X > x) = \Pr(X > y)$$

holds, then X is distributed according to a memoryless distribution. As the exponential distribution is the only memoryless distribution, X is indeed exponentially distributed.

The minimum of two independent exponentially distributed random variables X_1, X_2 with rates λ_1 and λ_2, respectively, is exponentially distributed with rate $\lambda_1 + \lambda_2$. Neither the maximum nor the sum of two exponentially distributed random variables are randomly distributed.

The expected value of an exponential distribution with rate λ is given by $\frac{1}{\lambda}$ as

$$\mathbb{E}(X) = \int_{-\infty}^{\infty} x\lambda e^{-\lambda x} \cdot u(x) dx = \lambda \int_{0}^{\infty} x e^{-\lambda x} dx = \lambda \cdot \frac{1}{\lambda^2} = \frac{1}{\lambda}$$

Remark 1. In the remainder of this thesis, we furthermore use the notion of *extended random variables*. This are continuous random variables whose codomain consists of the extended reals, i.e. $\mathbb{R} \cup \{-\infty, \infty\}$. Formal coverage of this is found in e.g. [AD00].

2.2. Markov automata

We briefly present Markov automata as a model and some relevant objectives on Markov automata. Markov automata were first introduced by Eisentraut *et al.* in [EHZ10a; EHZ10b]. The section is largely based on [Tim13], where a detailed discussion can be found.

2.2.1. Model definition

Definition 2.6 (MA). A *Markov automaton* (*MA*) $\mathcal{M}$ is a tuple $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ where

- S is a finite set of *states*,
- $\iota \in S$ is an *initial* state,
- Act is a set of *actions*,
- $\hookrightarrow \subseteq S \times \text{Act} \times \text{Distr}(S)$ is a set of *immediate transitions*.
- $\dashrightarrow \subseteq S \times \mathbb{R}_{>0} \times S$ is a set of *Markovian transitions*.
- AP is a set of *atomic propositions*.
- Lab is the *labelling* of the states, $\text{Lab}\colon S \to \mathcal{P}(\text{AP})$.

■

We explain the model using an adaption of the coffee-machine example[1].

Example 2.1. Consider a coffee machine at a CS department. We depict the Markov automaton for this machine in Figure 2.1.

The coffee machine is used by staff and by students. Staff members arrive at the coffee machine at a fixed rate of 5 members/hour, whereas students arrive at the machine at a fixed rate of 3 students/hour. At the machine, a user can either have coffee or espresso. Staff members always want espresso (we), whereas students non-deterministically want coffee (wc) or want espresso. Whereas staff members (who want espresso) always select espresso (se) from the machine and students who want coffee always select coffee (sc), students who want espresso are sometimes, with a chance of 0.1, too sleepy and select coffee. After the user has selected its choice, he or she get coffee (gc) or espresso (ge) based on their choice.

In state s_0, no user is at the coffee machine. Based on the respective rates, either a staff member or a student arrives at the machine (s_1 or s_2). The user wants coffee or wants espresso and with the given discrete probability selects espresso or coffee (s_3 or s_4). The user then gets the selected

[1] Which is found in e.g. [Tim13] and [BK08]

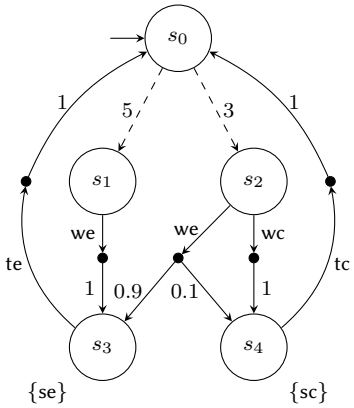


Figure 2.1.: A Markov automaton depicted.

product and the automaton return to the initial state. Notice that in this model, the selection and preparation is instantaneous, which also describes why no queue is modelled here.

Formally, the depicted Markov automaton is given by $(S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ with

- $S = \{s_0 \dots s_4\}$,
- $\iota = s_0$,
- $\text{Act} = \{\text{we}, \text{wc}, \text{te}, \text{tc}\}$,
- $\hookrightarrow = \{(s_1, \text{we}, \mathbb{1}_{s_3}), (s_2, \text{we}, \mu), (s_2, \text{wc}, \mathbb{1}_{s_4}), (s_3, \text{te}, \mathbb{1}_{s_0}), (s_4, \text{tc}, \mathbb{1}_{s_0})\}$ with $\mu(s_3) = 0.9$ and $\mu(s_4) = 0.1$,
- $\dashrightarrow = \{(s_0, 5, s_1), (s_0, 3, s_2)\}$,
- $\text{AP} = \{\text{se}, \text{sc}\}$,
- $\text{Lab}(s_3) = \{\text{se}\}$, $\text{Lab}(s_4) = \{\text{sc}\}$ and $\text{Lab}(s_i) = \emptyset$ for $0 \leq i \leq 2$.

▲

We introduce some auxiliary notation from Sazonov in [Saz14]. For each state $s \in S$, we define

- $\text{IT}(s) \subseteq \hookrightarrow$ as the set of outgoing immediate transitions, i.e.

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

- $\text{MT}(s) \subseteq \dashrightarrow$ as the set of outgoing Markovian transitions, i.e.

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

- $\text{act}(s) \subseteq \text{Act}$ as the set of action-labels on outgoing transitions, i.e.

$$\text{act}(s) = \{a \in \text{Act} \mid \exists \mu \in \text{Distr}(S)\ (s, a, \mu) \in \text{IT}(s)\}.$$

Immediate transitions are called *interactive probabilistic transitions* in [Tim13]. We partition the states of an MA into

- *interactive states* $\text{IS}_{\mathcal{M}}$, the set of states with at least one outgoing immediate transition, formally

$$\text{IS}_{\mathcal{M}}\{s \in S \mid \text{IT}(s) \neq \emptyset\},$$

- *deadlock states* $\text{DS}_{\mathcal{M}}$, the set of states without any outgoing transitions, formally

$$\text{DS}_{\mathcal{M}} = \{s \in S \mid \text{IT}(s) = \emptyset \wedge \text{MT}(s) = \emptyset\},$$

- *Markovian states* $\text{MS}_{\mathcal{M}}$, all other states, i.e. the set of states with Markovian transitions but without outgoing immediate transitions, formally

$$\text{MS}_{\mathcal{M}} = \{s \in S \mid \text{IT}(s) = \emptyset \wedge \text{MT}(s) \neq \emptyset\}.$$

As we do not synchronise multiple automata (*closed-world assumption*), we can assume w.l.o.g. that the Markov automaton is *action-deterministic*. An MA is action-deterministic if for each state, each outgoing transition has a unique action, or formally

$$|\mathrm{IT}(s)| = |\{a \in \mathrm{Act} \mid \exists \mu \in \mathrm{Distr}(S).\ (s, a, \mu) \in \mathrm{IT}(s)\}|$$

We use the *maximal progress assumption*, which —when combined with the closed world assumption— states that whenever a state has both outgoing immediate and Markovian transitions, then the Markovian transitions are never taken (as the probability that they're taken before any of the immediate transitions is taken is zero). As a direct consequence, we assume that interactive states have no outgoing Markovian transitions, formally

$$\forall s \in S\ \mathrm{IT}(s) \neq \emptyset \implies \mathrm{MT}(s) = \emptyset.$$

For a Markovian state s, we define the *exit rate of* s, $\mathrm{rate}(s)$, as the sum of all rates of outgoing transitions,

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

If $|\mathrm{MT}(s)| > 1$, multiple transitions "race" against each other, i.e. each transition fires after a delay, which is governed by the exponential distribution. The first transition to fire is taken. We say that this transition has *won* the *Markovian race*. Thus, the minimum of the delays governed by a exponential distribution is taken. Therefore, the minimum of the delays is a exponential distribution with a rate equal to the exit rate of s. We define for each Markovion state s, the probability distribution $\mathrm{next}(s) \in \mathrm{Distr}(S)$ with

$$s' \mapsto \frac{p}{\mathrm{rate}(s)} \text{ s.t. } p = \sum_{(s,\lambda,s') \in \dashrightarrow} \lambda.$$

By reviewing the exponential distribution, we obtain that $\mathrm{next}(s)(s')$ defines the probability that s' is the successor state of s.

We define paths through an MA partly following Guck *et al.* in [Guc12; GHHK+13]. In order to have a more uniform treatment, we introduce the extended action-set.

Definition 2.7. Let $\mathcal{M} = (S, \iota, \mathrm{Act}, \hookrightarrow, \dashrightarrow, \mathrm{AP}, \mathrm{Lab})$ be an MA. We define $\mathrm{Act}^\chi = \mathrm{Act} \cup \{\chi(r) \mid \exists r \exists s \in \mathrm{MS}\ r = \mathrm{rate}(s)\}$ as the set of actions extended by the rates found in the Markovian transitions. ■

With this notion, we define extended transitions.

Definition 2.8. Let $\mathcal{M} = (S, \iota, \mathrm{Act}, \hookrightarrow, \dashrightarrow, \mathrm{AP}, \mathrm{Lab})$ be an MA with $s, s' \in S$. We define the set of *extended transitions* $\to \subseteq S \times \mathrm{Act} \times \mathrm{Distr}(S)$ as follows.

$$\to = \{(s, a, \mu) \mid (s, a, \mu) \in \hookrightarrow \vee (\mathrm{rate}(s) > 0 \wedge a = \chi(\mathrm{rate}(s)) \wedge \mu = \mathrm{next}(s))\}$$ ■

A Markovian state has now exactly one outgoing transition. Sometimes, we are not interested in the actual source state of the transition. The set of *outgoing connections from* s, written $\mathrm{outgoing}(s)$ is defined as $\{(a, \mu) \mid (s, a, \mu) \in \to\}$.

We define paths using the extended transitions.

Definition 2.9 (Path in MA)**.** Let $\mathcal{M} = (S, \iota, \mathrm{Act}, \hookrightarrow, \dashrightarrow, \mathrm{AP}, \mathrm{Lab})$ be an MA. A *path* π *in* $\mathcal{M}$ is either an infinite tuple

$$\pi = s_0 a_1 \mu_1 t_1 s_1 a_2 \mu_2 t_2 s_2 \ldots \in S \times \mathrm{Act}^\chi \times \mathrm{Distr}(S) \times \mathbb{R}_{\geq 0} \times S)^\omega,$$

or a finite tuple with $n \in \mathbb{N}$

$$\pi = s_0 a_1 \mu_1 t_1 s_1 \ldots a_n \mu_n t_n s_n \in S \times \mathrm{Act}^\chi \times \mathrm{Distr}(S) \times \mathbb{R}_{\geq 0} \times S)^n,$$

such that for all $0 \leq i\ (< n)$ $s_i, s_{i+1} \in S$ and $(s_i, a_i, \mu_i) \in \to$, $\mu_i(s_{i+1}) > 0$ and $t_i = 0 \iff$

$a_i \in \text{Act}$. If π is finite, then the path is said to be finite and its length is defined as n, and infinite otherwise. ■

We analogously define paths without timing information.

Definition 2.10 (Time-abstract path)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA. A *time-abstract path π in $\mathcal{M}$* is either an infinite tuple

$$\pi = s_0 a_1 \mu_1 s_1 a_2 \mu_2 s_2 \ldots \in S \times \text{Act}^\chi \times \text{Distr}(S) \times S)^\omega,$$

or a finite tuple with $n \in \mathbb{N}$

$$\pi = s_0 a_1 \mu_1 s_1 \ldots a_n \mu_n s_n \in S \times \text{Act}^\chi \times \text{Distr}(S) \times S)^n,$$

such that $\forall 0 \leq i (< n)\ \exists t_i \in \mathbb{R}$ such that $s_0 a_1 \mu_1 t_1 s_1 a_2 \mu_2 t_2 s_2 \ldots (a_n \mu_n t_n s_n)$ is a (infinite) path. ■

The set of all finite paths in $\mathcal{M}$ starting from $s \in S$ is denoted $\text{FinPaths}_\mathcal{M}(s)$. The set of infinite paths is denoted $\text{Paths}_\mathcal{M}(s)$. Given a path π, the prefix of length n is given as $\text{pre}_n(\pi)$. We drop the reference to the MA in all concepts above whenever it is clear from the context. We furthermore omit (ι) whenever we refer to states from the initial state. A finite path $\pi = s_0 a_1 \mu_1 s_1 \ldots s_n$ with $n \geq 0$ is called *immediate* if $\forall i < n$, $a_i \in \text{Act}$.

Definition 2.11 (Elapsed time)**.** Let $\mathcal{M}$ be an MA and $\pi \in \text{FinPaths}_\mathcal{M}$. The *elapsed time* of π is the sum of all sojourn times on the path, i.e. we define $\text{elapsed}(\pi) = \sum_{i=1}^{|\pi|} \pi_{4 \cdot i}$ ■

Definition 2.12 (Zeno path)**.** Let $\mathcal{M}$ be an MA and $\pi \in \text{Paths}_\mathcal{M}$. The infinite path π is called *Zeno*, if

$$\lim_{n \to \infty} \text{elapsed}(\text{pre}_n(\pi)) < \infty$$ ■

An MA $\mathcal{M}$ is called *Zeno-free* if there exist no Zeno path in $\mathcal{M}$ which starts in ι. A Markov-automaton is Zeno-free if and only if all cycles of reachable states contain at least one *Markovian state*. In this thesis, we assume that all Markov automata are Zeno-free.

Furthermore, we assume each Markov automaton $\mathcal{M}$ to be *deadlock-free*, i.e. $\text{DS}_\mathcal{M} = \emptyset$. Any Markov automaton $\mathcal{M}$ with $\text{DS}_\mathcal{M} \neq \emptyset$ is transformed in a deadlock-free automaton by adding Markovian self-loops to each deadlock state.

Remark 2. Adding immediate transitions instead of Markovian transitions adds Zeno-paths to the model.

In the interactive states, non-determinism occurs whenever multiple outgoing transitions are present. To resolve this non-determinism, we introduce the concept of *schedulers* (also called *policies* or *strategies* in literature).

Definition 2.13 (General scheduler)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA. A measurable[1] function $\mathcal{S} \colon \text{FinPaths}_\mathcal{M} \to \text{Distr}(\text{Act}) \cup \{\perp\}$ such that for a finite path $\pi \in \text{FinPaths}_\mathcal{M}$,

$$\mathcal{S}(\pi) = \perp \text{ if } \pi_\downarrow \in \text{MS}_\mathcal{M}$$

and

$$\text{supp}(\mathcal{S}(\pi)) \subseteq \text{act}(\pi_\downarrow) \text{ otherwise,}$$

is called a *general (measurable) scheduler on $\mathcal{M}$*. The set of all general schedulers is denoted $\text{GMSched}_\mathcal{M}$. ■

Different schedulers are discussed by Neuhäuser *et al.* in [NSK09].

For this thesis, we furthermore consider time-homogeneous schedulers.

[1] measurable is property from measure theory, cf. [AD00]. In the context of the thesis, it is not of particular importance.

Definition 2.14 (Time-homogeneous scheduler)**.** Let $\mathcal{M}$ be an MA and $\mathcal{S}$ a scheduler over $\mathcal{M}$. If for all $\pi, \pi' \in \text{FinPaths}_{\mathcal{M}}$ it holds that

$$\text{tAbs}(\pi) = \text{tAbs}(\pi') \implies \mathcal{S}(\pi) = \mathcal{S}(\pi')$$

then $\mathcal{S}$ is *time-homogeneous*. ■

Please, notice that time-homogeneous schedulers are measurable, as discussed by Zhang and Neuhäuser in [ZN10].

A very simple form of scheduler, which is used throughout the thesis is the *stationary* scheduler.

Definition 2.15 (Stationary scheduler)**.** Let $\mathcal{M}$ be an MA and $\mathcal{S}$ a scheduler over $\mathcal{M}$. If for all $\pi, \pi' \in \text{FinPaths}_{\mathcal{M}}$ it holds that

$$\pi_{\downarrow} = \pi'_{\downarrow} \implies \mathcal{S}(\pi) = \mathcal{S}(\pi')$$

then $\mathcal{S}$ is called *stationary*. ■

Definition 2.16 (Deterministic scheduler)**.** Let $\mathcal{M}$ be an MA and $\mathcal{S}$ a measurable scheduler over $\mathcal{M}$. If

$$\text{Ran}(\mathcal{S}) \subseteq \{\mathbb{1}_a \mid a \in \text{Act}\} \cup \{\bot\}$$

then $\mathcal{S}$ is called *deterministic*. The set of deterministic schedulers is given as $\text{DSched}_{\mathcal{M}}$. ■

The set of *deterministic time-homogeneous* and *deterministic stationary schedulers* are denoted with DTHSched and DSSched, respectively. For a Markov automaton $\mathcal{M}$ with some state s and a scheduler $\mathcal{S}$, probability mass can be attached to each measurable set of infinite paths starting in ι via a cylinder set construction, cf. [Saz14]. We denote this as $\Pr_{\mathcal{S}}^{\mathcal{M},s}(\pi)$. We omit $\mathcal{M}$ whenever it is clear from the context and omit s whenever it is the initial of $\mathcal{M}$.

Other Markovian models Markov automata are a general model. In this thesis, three other models are used, which are restricted Markov automata.

Definition 2.17 (IMC)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be a Markov automaton. If $\hookrightarrow \subseteq S \times \text{Act} \times \{\mathbb{1}_s \mid s \in S\}$, then $\mathcal{M}$ is a *Interactive Markov Chain* (*IMC*). ■

IMCs are discussed in-depth by Hermanns in [Her02]. Model-checking of IMCs is discussed by Neuhäuser in [Neu10].

Definition 2.18 (CTMC)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be a Markov automaton. If $\hookrightarrow = \emptyset$, then $\mathcal{M}$ is a *Continuous Time Markov Chain* (*CTMC*). ■

CTMCs are discussed by Norris in [Nor98]. Model-checking on CTMCs is discussed by Baier *et al.* in [BHHK03].

Definition 2.19 (PA)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be a Markov automaton. If $\dashrightarrow = \emptyset$, then $\mathcal{M}$ is a *probabilistic automaton* (*PA*). ■

Remark 3. As we assume Markov automata to be action-deterministic, the probabilistic automata defined above are also *Markov Decision Processes* (MDP).

PAs are discussed by Stoelinga in [Sto02]. Model-checking MDPs is covered by [BK08].

Definition 2.20 (LTS)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be a Markov automaton. If $\dashrightarrow = \emptyset$ and $\hookrightarrow \subseteq S \times \text{Act} \times \{\mathbb{1}_s \mid s \in S\}$, then $\mathcal{M}$ is a *labeled transition system* (*LTS*). ■

Labeled transition systems are a qualitative model. Model checking them is a seperate research area, an introduction to the topic can be found in [BK08]. Labeled transition systems can be composed by smaller transition systems in several ways, see [BK08]. As IMCs and MAs are extensions of LTSs, they can be composed likewise, as discussed in [Tim13].

2.2.2. Quantitative objectives

In the context of this thesis, we are interested in four different objectives, *unbounded reachability*, *time-bounded reachability*, and (conditional) *expected time*.

In this section, we always assume a subset of the state space to be the *goal-set*. Often, such a goal set is described by (propositional) formula over the state-labelling.

Unbounded reachability Unbounded reachability is the probability to take a path through the MA in which eventually, a goal-state is visited.

Definition 2.21 (Unbounded reachability)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA and $G \subseteq S$ a set of *goal-states*. The random variable $H_G \colon \text{Paths} \to \{0, 1\}$ yields 1 if there exists a state from G on π, and 0 otherwise, formally

$$H^t_G(\pi) = 1 \iff \exists i \in \mathbb{N} \text{ s.t. } \text{pre}_i(\pi) \downarrow \in G.$$

The *(unbounded) reachability probability to reach G from s in $\mathcal{M}$ under scheduler $\mathcal{S}$*, denoted $\Pr^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond G)$ is given by

$$\Pr^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond G) = \int_{\text{Paths}} H_G(\pi) \Pr^{\mathcal{M},s}_{\mathcal{S}}(d\pi). \qquad \blacksquare$$

The minimal (maximal) reachability probability to reach G from s in $\mathcal{M}$ is obtained by taking the infimum (supremum) reachability probability over all schedulers. Based on results on MDPs (cf. [BK08]), we can deduce that it suffices to take the minimum (maximum) of ranging over the stationary deterministic schedulers. The minimal reachability probability is denoted $\Pr^{\mathcal{M}}_{\min}(s, \Diamond G)$, the maximum analogously. We omit $\mathcal{M}$ whenever it is clear from the context. We omit s whenever it corresponds to the initial state of $\mathcal{M}$.

Calculating the unbounded reachability can be done on the *embedded MDP* after replacing all Markovian transitions by interactive transitions, as timing is not important for unbounded reachability. For details, we refer to Hafeti and Hermanns [HH12].

Time-bounded reachability Time-bounded reachability is the probability to take a path through the MA in which eventually, but before the elapsed time passes a given bound, a goal-state is visited.

Definition 2.22 (Time-bounded reachability)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA, $G \subseteq S$ a set of *goal-states* and $t \in \mathbb{R}_{\geq 0}$ a *deadline*. The random variable $H^t_G \colon \text{Paths} \to \{0, 1\}$ yields 1 if there exists a prefix of π with a state in G and an elapsed time $< t$, and 0 otherwise, formally

$$H^t_G(\pi) = 1 \iff \exists i \in \mathbb{N} \text{ s.t. } \text{pre}_i(\pi) \downarrow \in G \wedge \text{elapsed}(\text{pre}_i) < t.$$

The *reachability probability to reach G from s within t in $\mathcal{M}$ under scheduler $\mathcal{S}$*, denoted $\Pr^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond^{\leq t} G)$ is given by

$$\Pr^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond^{\leq t} G) = \int_{\text{Paths}} H^t_G(\pi) \Pr^{\mathcal{M},s}_{\mathcal{S}}(d\pi). \qquad \blacksquare$$

The minimal (maximal) reachability probability to reach G from s in $\mathcal{M}$ is obtained by taking the infimum (supremum) reachability probability over all schedulers. Based on results on IMCs (cf. [NSK09]), we can deduce that it does not suffice to restrict the schedulers to time-homogeneous schedulers. The minimal reachability probability is denoted $\Pr^{\mathcal{M}}_{\min}(s, \Diamond^{\leq t} G)$, the maximum analogously. We omit $\mathcal{M}$ whenever it is clear from the context. We omit s whenever it corresponds to the initial state of $\mathcal{M}$.

The computation of time-bounded reachability be done via a digitisation approach which partitions the time-interval into smaller intervals and a generalised form of uniformisation to calculate an MDP, as explained in [GHHK+13].

Expected Time The results presented here are taken from [Guc12]. The expected time to reach a set of states G is the amount of time that we can expect the MA take before reaching a state in G. It is the (probability) weighted average of the elapsed times before we visit a goal state.

Definition 2.23 (Expected time)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA and $G \subseteq S$ a set of *goal-states*. The (extended) random variable $V_G \colon \text{Paths} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ yields the first time-point t such that π contains a state in G, formally

$$V_G(\pi) = \min\{\text{elapsed}(\text{pre}_n(\pi)) | \pi_\downarrow \in G\}$$

with $\min \emptyset = \infty$. The *expected time to reach G from s in $\mathcal{M}$ under scheduler $\mathcal{S}$*, denoted $\mathsf{ET}_{\mathcal{S}}^{\mathcal{M}}(s, \Diamond G)$ is given by

$$\mathsf{ET}_{\mathcal{S}}^{\mathcal{M}}(s, \Diamond G) = \mathbb{E}_{\mathcal{S}}^{\mathcal{M},s}(V_G) = \int_{\text{Paths}} V_G(\pi) \text{Pr}_{\mathcal{S}}^{\mathcal{M},s}(d\pi). \qquad \blacksquare$$

The *minimal expected time to reach G in $\mathcal{M}$* is obtained by taking the infimum over all schedulers on $\mathcal{M}$. The *maximal expected time* is obtained by taking the supremum. In [Guc12], it is shown that it suffices to take the minimum (maximum) over all deterministic stationary schedulers to obtain the minimal (maximal) expected time. The minimal expected time to reach G from s is denoted $\mathsf{ET}_{\min}^{\mathcal{M}}(s, \Diamond G)$. The maximal analogously. Again, we drop $\mathcal{M}$ and s whenever it is clear from the context, or when s is the initial state of $\mathcal{M}$. Expected time can be calculated via a reduction to a *stochastic shortest path problem*, which can be encoded as a linear program.

We see that the integral does not necessarily exists, as there might be paths which never visit a goal state. Therefore, we consider a conditional expected time property, inspired by Baier *et al.* [BKKM14].

Definition 2.24. Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA and $G \subseteq S$ a set of *goal-states* and G' an *assumed visited* set. Let the random variables $H_{G'}$, V_G be defined as in Definition 2.2.2 and Definition 2.23, respectively. The *conditional expected time to reach G from s in $\mathcal{M}$ under the assumption that eventually G' is visited*, denoted $\mathsf{ET}_{\mathcal{S}}^{\mathcal{M}}(s, \Diamond G | \Diamond G')$ is given by

$$\mathsf{ET}_{\mathcal{S}}^{\mathcal{M}}(s, \Diamond G | \Diamond G') = \mathbb{E}_{\mathcal{S}}^{\mathcal{M},s}(H_{G'} \cdot V_G) = \int_{\text{Paths}} H_{G'}(\pi) \cdot V_G(\pi) \text{Pr}_{\mathcal{S}}^{\mathcal{M},s}(d\pi). \qquad \blacksquare$$

Assumption 1. *In the remainder, we assume that the set of goal states G is given by $\{s \in S \mid Lab(s) = a\}$ for some atomic proposition a.*

2.2.3. Equivalence relations

In order to compare two different Markov automata, it is important to define a notion of equivalence. Many different forms of equivalence have been proposed. We only give the relevant notions following [Tim13] which contains a good overview. A more extensive treatment of weak and strong bisimulation on MAs is given in [Saz14].

A very strict equivalence relation is *isomorphism*. Intuitively, two Markov automata are isomorph if one can obtain the other by renaming the reachable states.

Definition 2.25. Given an MA $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ with two states $s, t \in S$, s and t are *isomorphic* if there exists a bijection $f \colon S \to S$ such that $f(s) = t$ and

$$\forall S' \in S\, a \in \text{Act}^\chi, \mu \in \text{Distr}(S).\, \text{Lab}(f(s')) = \text{Lab}(s') \wedge (s', a, \mu) \in \hookrightarrow \iff (f(s'), a, \mu_f) \in \hookrightarrow$$

where μ_f is given by $\mu_f(x) = \mu(f^{-1}(x))$ for all $x \in S$. $\blacksquare$

Given two isomorph MAs, all earlier defined measures coincide [Tim13].

Strong bisimulation Strong bisimulation puts states into equivalence classes where each taken transition from a state in a given equivalence class can be mimicked from another state in the same equivalence class.

Definition 2.26. Given an MA $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$. An equivalence relation $R \subseteq S \times S$ is a *strong bisimulation for* $\mathcal{M}$ if for all $(s, s') \in R$ and for all $a \in \text{Act}^{\chi} \setminus \text{Act}$ the following holds:

- $\text{Lab}(s) = \text{Lab}(s')$
- $(s, a, \mu) \in \hookrightarrow \implies \exists \mu' \in \text{Distr}(S).\ (s', a, \mu') \in \hookrightarrow \wedge \mu \equiv_R \mu'$.

and for all $a \in \text{Act}$

- $\text{Lab}(s) = \text{Lab}(s')$
- $(s, a, \mu) \in \hookrightarrow \implies \exists \mu' \in \text{Distr}(S).\ \exists a' \in \text{Act}(s', a', \mu') \in \hookrightarrow \wedge \mu \equiv_R \mu'$.

Two states $s, s' \in S$ are strongly bisimilar if there exists a strong bisimulation R for $\mathcal{M}$ such that $(s, s') \in R$. We write $s \approx_s s'$. Two MAs are strongly bisimilar if their initial states are strongly bisimilar in the union[1] of the Markov automata. ■

Notice that we allow for different immediate action-labels, as we assume a closed world and do not derive any measures from the actions chosen.

Given two strong bisimilar MAs, time-unbounded reachability and expected time coincide [Saz14]. If the MAs are IMCs, then also time-bounded reachability coincides [HK10].

Weak bisimulation We consider an instance of weak bisimulation. We restrict ourselves to IMCs here, as this notably simplifies the presentation. Furthermore, we restrict ourselves to a very coarse instance as this suffices for our needs.

The idea behind weak bisimulation is that we consider paths of transitions where it is not observable whether a transition has occurred. In this setting, we cannot observe paths who do not change their labelling. We do, however, observe Markovian transitions.

Therefore, we consider a sequence of an immediate path, a Markovian transition and an immediate path. We formalise this as follows.

Definition 2.27 (Weak transition)**.** Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an MA with $s, s' \in S$. If either there exists an immediate path π from s to s' with $\forall y, y' \in \pi \cap S$ with $\text{Lab}(y) = \text{Lab}(y')$. ■

Definition 2.28. Let $\mathcal{M} = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be an IMC. An equivalence relation $R \subseteq S \times S$ is a *weak bisimulation for* $\mathcal{M}$ if for all $(s, s') \in R$ either $s \approx_s s'$ or

- $\text{Lab}(s) = \text{Lab}(s')$, and
- $\forall t$ s.t. there exists a weak transition from s to t, $\exists t'$ s.t. $(t, t') \in R$ and there exists a weak transition from s' to t'.

Two states $s, s' \in S$ are *weakly bisimilar* if there exists a weak bisimulation R for $\mathcal{M}$ such that $(s, s') \in R$. We write $s \approx_w s'$. Two MAs are weakly bisimilar if their initial states are weakly bisimilar in the union of the Markov automata. ■

We notice that two states which are weakly bisimilar according to the definition above are branching bisimilar according to [Tim13][Definition 3.31].

For acyclic IMCs, this notion of weak bisimulation thus preserves expected time and time-unbounded reachability. Based on statements from [Tim13] and [Saz14], we conjecture that this notion also preserves time-bounded reachability on acyclic IMCs (for cyclic IMCs, this does not hold in general due to a notion called divergence, cf. [Her02]).

2.3. Graph Rewriting

In this section, we introduce the required background in *graph rewriting*. The overall discussion follows Zambon [Zam13]. We give a theoretic background in algebraic graph rewriting by DPO and a introduction to Groove, which is the environment we use to define our rewrite system. For a more complete coverage of the topic, we refer to Rozenbe.g. [Roz97]. Tool-support for model transformation in general, and graph transformation in particular, is found in numerous tools, cf. Jakumeit *et al.* in [JBWD+14].

Graph rewriting, also called *graph transformation* is a method to modify an input graph to other graphs by the sequential application of a given set of rules, called a *graph grammar*.

[1] The union of two MA is the disjoint union of their state spaces, see [Tim13]

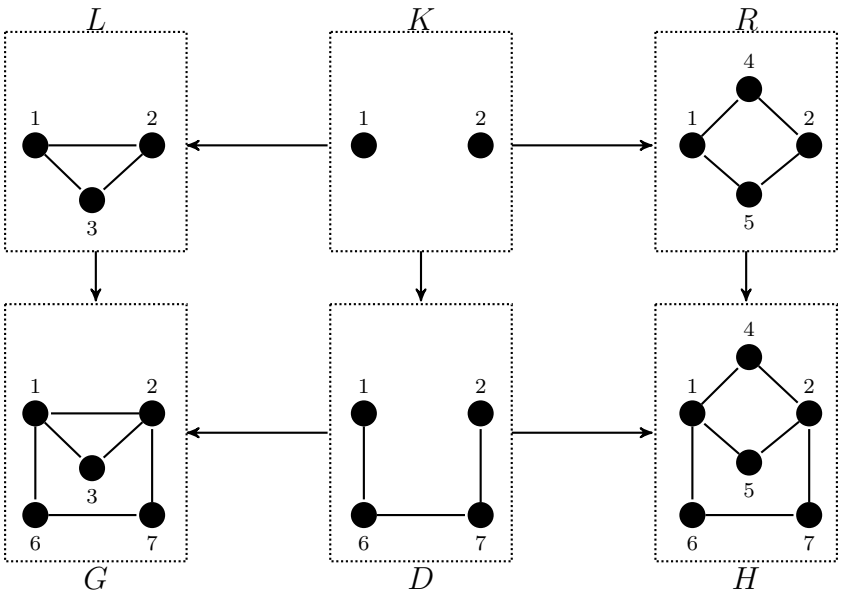


Figure 2.2.: Example for DPO graph rewriting [EEPT06]

Different approaches to graph transformation are found in the literature, e.g. *hyper-e.g. replacement grammars*, cf. Drewes *et al.* [DKH97], and *term graph replacement*, cf. Plump [Plu02]. In this thesis, we use the algebraic approach which has its roots in category theory. Multiple variants for this algebraic approach exist. We choose the double pushout (DPO) approach, which is described by Ehrig in [Ehr79]. In most approaches, including DPO, a rule entails two graphs L and R. Rewriting of a host graph G is done by finding a subgraph in G which matches L, e.g. as there is some mapping from L to the subgraph in G. This subgraph is then replaced by R. Notice that a particular The effect of multiple rules in a grammar which might be applicable to the same host graph is discussed in the discussion of Groove.

Before we give the formal definition of rewriting, we use an example from [EEPT06] to depict the procedure.

Example 2.2. We describe the application of a given rewrite rule on a given host graph. All graphs have a vertex label set $\{1 \dots 7\}$ and an e.g. label set $\{x\}$. We depict the rule (graphs L, K, R) and the host-graph (G) as well as an intermediate step (D) and the final result (H) in Figure 2.2. We depict vertex labels, but omit the e.g. labels. The arrows between the graphs depict the graph morphisms and are explained later in the section.

We consider a rule given by the three graphs L, K and R. Graph L describes the subgraph which has to be matched in a host graph. The graph K describes a subgraph of L which we call the *interface*. After successfully matching L in G, elements which are matched L but not the interface are removed, yielding graph D. D has to be a valid graph, so dangling edges (i.e. edges without a source or target vertex) in D are not allowed. To prevent dangling edges in D, additional conditions on the match of L in G are put. In this particular example, we trivially match L in G, after which we remove all matched edges and the vertex labelled 3. This yields D. Notice that we do not have dangling edges in D, as all edges which lead to a deleted vertex are also removed.

To finalise the rewriting step, graphs R and D are merged, such that elements which originate from the interface are not duplicated. In this particular example, we add graph R to graph D, such that the two vertices from K are merged, yielding H. ▲

2.3.1. Theory

We apply graph rewriting on *labelled graphs*, that is directed graphs with labelled vertices and labelled edges.

Definition 2.29 (Labelled Graph)**.** Let $\Sigma = (\Sigma_v, \Sigma_e)$ a *the label set* with Σ_v a set of *vertex labels* and Σ_e a set of *e.g. labels*. A *labelled graph* G is a tuple $G = (V, E, l)$ with a finite set of *vertices*

V and a set of *edges* $E \subseteq V \times V$. Moreover $l = (\mathrm{lv}, \mathrm{le})$ is the *labelling* with $\mathrm{lv} : V \to \Sigma_v$ and $\mathrm{le} : E \to \Sigma_e$. ■

We use $\mathfrak{G}(\Sigma)$ to denote the set of all labelled graphs over Σ. It is helpful to have a functional view on edges.

Definition 2.30 (Source/target functions)**.** Given a graph $G = (V, E, l)$. The *source function* $s_G \colon E \to V$ is defined as $s_G \colon (s,t) \mapsto s$. The *target function* $t_G \colon E \to V$ is defined as $t_G \colon (s,t) \mapsto t$. ■

The following formalises the notions of incoming (outgoing) edges, as well their source (target) vertices.

Definition 2.31. Given a graph $G = (V, E, l)$. We define $\mathrm{In} : V \to \mathcal{P}(E)$ as $v \mapsto \{e \in E \mid t_G(e) = v\}$ and $\mathrm{InV} \colon V \to \mathcal{P}(V)$ as $v \mapsto \{s_G(e) \in V \mid e \in \mathrm{In}(v)\}$.

Analogously, we define $\mathrm{Out} : V \to \mathcal{P}(E)$ as $v \mapsto \{e \in E \mid s_G(e) = v\}$ and $\mathrm{OutV} \colon V \to \mathcal{P}(V)$ as $v \mapsto \{t_G(e) \in V \mid e \in \mathrm{Out}(v)\}$. ■

Morphisms are structure-preserving mappings. Graph morphisms map adjacent vertices to adjacent vertices and preserve labelling of labels and edges.

Definition 2.32 (Graph morphism)**.** Given two graphs $G = (V_G, E_G, (\mathrm{lv}_G, \mathrm{le}_G))$ and $H = (V_H, E_H, (\mathrm{lv}_H, \mathrm{le}_H))$. Then $g \colon G \to H$ with $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ is a *graph morphism* if it

- preserves sources $s_H \circ g_E = g_V \circ s_G$, and
- preserves targets $t_H \circ g_E = g_V \circ t_G$, and
- preserves vertex labels $\mathrm{lv}_H \circ g_V = \mathrm{lv}_G$, and
- preserves e.g. labels $\mathrm{le}_H \circ g_E = \mathrm{le}_G$.

■

Remark 4. In many cases, we identify an arbitrary morphism between two graphs G and H by $G \to H$. That doesn't mean that there is a unique morphism. We refrain from this short form if multiple morphism between G and H are considered.

A graph morphism g is injective (surjective) if g_V and g_E are injective (surjective). It helpful to embed vertices and edges in other nodes. An embedding is an injective morphism s.t. $g_V(v) = v$ and $g_E(e) = e$ for all $v \in V_G$ and $e \in E_G$. The composition of two morphisms $G \to H$ and $H \to K$ is a morphism, denoted $G \to H \to K$.

The concept of a *pushout* helps to formalise merging two graphs. The construction is illustrated in Figure 2.3.

Definition 2.33 (Pushout)**.** Given three graphs A, B, C and graph morphisms $A \to B$ and $A \to C$. Then $(D, B \to D, C \to D)$ is a *pushout* if

- $A \to B \to D = A \to C \to D$.
- For all graphs D' and graph morphisms $B \to D', C \to D'$ s.t.

  $$A \to B \to D' = A \to C \to D'$$

  there exists a unique graph morphism $D \to D'$ s.t.

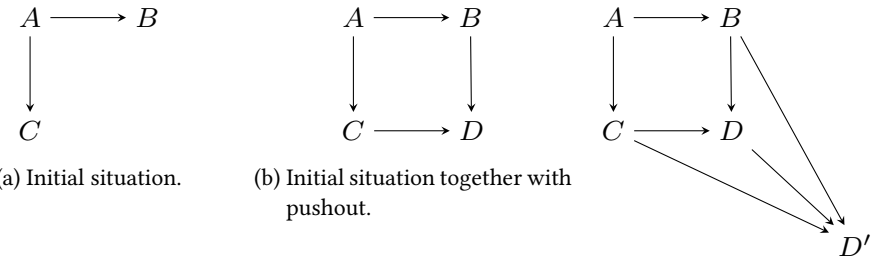
  $$B \to D \to D' = B \to D', \quad C \to D \to D' = C \to D'.$$

We denote the pushout with $ABCD$ or equivalently $ACBD$. ■

The following lemma sketches the construction of a pushout.

Lemma 2.1 (Pushout construction)**.** *[Ehr79] Given graphs A, B, C with graph morphisms $b : A \to B$ and $c : A \to C$.*

Let $\sim_V$ be a relation such that $b_V(v) \sim c_V(v)$ for all $v \in V_A$. We define the equivalence relations $\approx_V$ as the smallest equivalence relation containing $\sim_V$.



(a) Initial situation.

(b) Initial situation together with pushout.

(c) Pushout criterion depicted.

Figure 2.3.: Pushout construction.

Let $D = (V_D, E_D, l_D)$. *We have that* $V_D = V_B \cup V_C / \approx$ *and* $([s],[t]) \in E_D$ *iff* $\exists v \in [s], v' \in [t].\ (v,v') \in E_B \cup E_C$. *Moreover* $l_D = (lv_D, le_D)$ *is given by*

$$lv_D([v]) = \begin{cases} lv_B([v]) & v \in V_B \\ lv_C([v]) & v \in V_C \end{cases}$$

$$le_D(([s],[t])) = \begin{cases} le_B(([s],[t])) & \exists v \in [s], v' \in [t].\ (v,v') \in E_B \\ le_C(([s],[t])) & \exists v \in [s], v' \in [t].\ (v,v') \in E_C \end{cases}$$

The morphism $f : B \to D$ *is given by* $f_V(v) = [v]$ *and* $f_E((s,t)) = ([s],[t])$. *The morphism* $C \to D$ *is defined analogously.*

We now formalise graph rewriting by DPO. We start by formalising a rewrite rule (the upper layer in Figure 2.2 on page 17).

Definition 2.34 (Rewrite rule for graphs)**.** Given three graphs L, K, R. We call a tuple $r = (L, K, R, K \to L, K \to R)$ a *graph rewrite rule* if $K \to L$ is an inclusion. We call L the *left-hand side of* r, R the *right-hand side of* r and K the *interface of* r. ■

We often abbreviate a rule with $(L \leftarrow K \to R)$. A rule is injective if $K \to R$ is.

Applying a rewrite rule on a graph is called a rewrite step. We first define such a step, and then give a lemma which gives a constructive description of such a rewrite step and the obtained result.

Definition 2.35 (Rewrite step)**.** Given two graphs G and H and a rewrite rule $r = (L \leftarrow K \to R)$, the *host graph* G can be rewritten to H by r if there exists a graph D such that $KLDG$ and $KRDH$ are pushouts. We write $G \xrightarrow{r} H$ and call this a *rewrite step on* G *by* r. We call r *applicable on* G. ■

Lemma 2.2. *[Ehr79] Given a rule* $r = (L \leftarrow K \to R)$ *and a graph* G, *the graph* H *as in Definition 2.35 can be constructed as follows.*

1. *Find a graph morphism* $\kappa : L \to G$. *Check that the following conditions hold.*
   - *For all* $(s,t) \in E_G \setminus \kappa_E(L)$, *it holds that* $s, t \notin \kappa_G(L) \setminus \kappa_G(K)$ (dangling e.g. condition).
   - *For all* $\{v, v'\} \subseteq V_L$ *and for all* $\{e, e'\} \in E_L$, $\kappa(v) = \kappa(v') \implies \{v, v'\} \subseteq V_K$ *and* $\kappa(e) = \kappa(e') \implies \{e, e'\} \subseteq E_K$ (identification condition)..
2. *Get* $D = (V_G \setminus (\kappa_V(L) \setminus \kappa_V(K)), E_G \setminus (\kappa_E(L) \setminus \kappa_E(K)), l_D)$ *where* $l_D = (lv_G|V_D, le_G|E_D)$. *Get the graph morphism* $K \to D$ *as* $\kappa|K$, *and the embedding* $D \to G$.
3. *Construct the pushout* $KRDH$ *by using Lemma 2.1*

2.3.2. Groove

In this section, we consider Groove [GdRZ+12], a general-purpose labelled graph rewriting tool set. The description of Groove in this section is based on [Zam13]. A complete description of features is given in the Grooveuser manual [RBKS12]. We illustrate the usage of Groove by a running example.

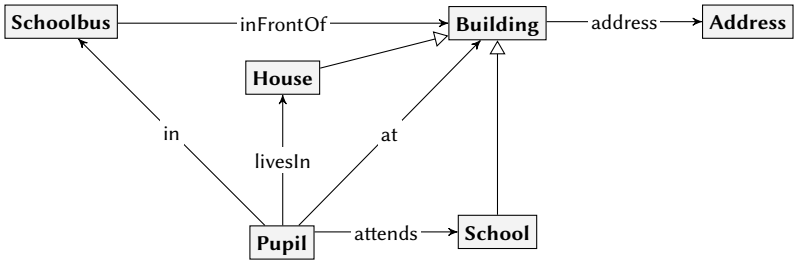


Figure 2.4.: The type declaration for the school bus example.

Remark 5. Groove internally uses an alternative algebraic approach for graph rewriting, called *single pushout* (SPO). SPO can simulate any rewrite grammar given in DPO [Ehr79], thus a rewrite grammar given in DPO can be defined inside Groove. In the remainder of the thesis, we only require basic knowle.g. of DPO and Groove, as given in this chapter.

Amongst others, Groove contains a generator, which given an *input graph* and a set of *rewrite grammar*, recursively generates a state space of produced graphs. To this end, the generator starts with a state space of only the input graph. Now, in every step, Groove takes a graph G from the state space and a rule r from the grammar. If the rule can be applied to the graph, the state space is extended with the result of applying r to G. Moreover, Groove contains a simulator, which provides a GUI to manually execute the graph rewriting and to specify graphs and rules. Whenever we refer to Groove, we refer to the combination of simulator and generator.

In the remainder of this section, we briefly discuss how rewrite rules in Groove are specified, and how the recursive exploration of the state space can be guided. We illustrate these features by an running example.

2.3.2.1. Groove graph specification

Graphs are specified by a set of nodes and directed multi-edges between them. A node can have a type, which is a encoded as a node label. Furthermore, nodes can have identifiers (invisible to the grammar) and labels (labelled self-loops). Edges are all labelled by a set of labels. Multi-edges are equivalent to a single edges with the union of the labels. A *type-graph* is a graph which restrict the set of well-formed graphs. It consists of a node for each allowed type and labelled edges between these types to define the set of allowed edges in a graph. Type graphs support a kind of *inheritance* as also known in object oriented programming [Pie02]. A type T which inherits properties from another type T' may have, additionally to its own set of allowed in- and outgoing edges, the in- and outgoing edges of T'.

In this example, we introduce the general setting and deduce a type graph for the setting. We then consider a particular example and depict the graph for the scenario.

Example 2.3. We consider an (extraordinary) school bus which is used to bring pupils from their home to the their school. We consider a world with buildings, (school) busses and pupils. Houses and schools are buildings, and buildings have an address. A bus can be in front of any building. A pupil has a name. Their home is a house in which they live and each pupil attends a school. Moreover, a pupil can be at a building or in a bus. We depict a type graph for this in Example 2.3.

We consider a small example scenario in Section 2.3.2.1 containing only one school and one bus, and two pupils with their house Notice that we assume the pupils to be at their homes initially and that the bus is in front of one of the pupils homes. We use this small scenario in further examples. ▲

2.3.2.2. Groove rule specification

A rule in Groove is given by a large graph consisting of four different categories.

- *Reader elements* (continuous thin / black) are nodes or edges which have to be matched in order to make the rule applicable, but are untouched.

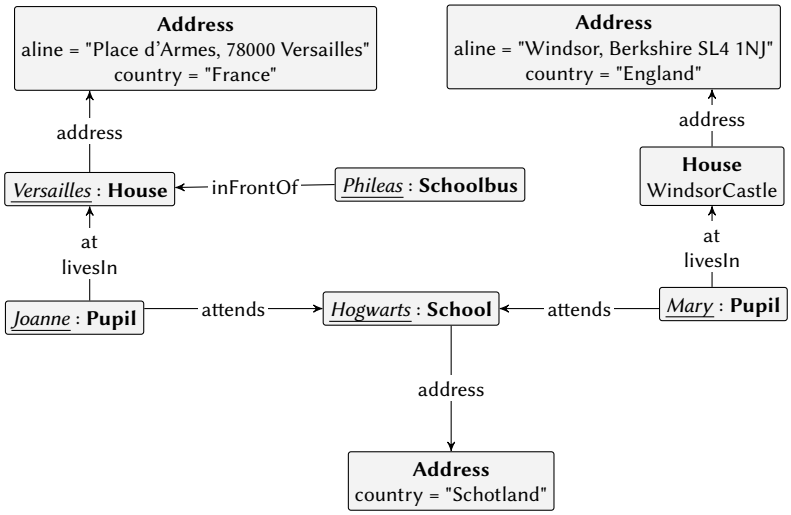


Figure 2.5.: The input graph for the school bus example.

- *Embargo elements* (dashed fat / red) are nodes or edges which make the rule inapplicable whenever they are successfully matched.
- *Eraser elements* (dashed thin / blue) are nodes or edges which have to be matched in order to make the rule applicable. These elements are removed during the application of the elements. Please, notice that the removal of a node als removes all adjacent edges.
- *Creator elements* (continuous fat / green) are nodes or edges which are added during the application of the rule.

Furthermore, nodes can be quantified and labels can be tested, e.g. whether a label "age" is greater than some number. Moreover, paths between two nodes can be described by a regular expression.

We illustrate this by extending our running example. For a more complete introduction, we refer to [RBKS12].

Example 2.4. We continue using the setting and scenario from Example 2.3.

Starting from the initial scenario, the bus has a selection of actions which can be executed sequentially. We encode these actions as rewrite rules to reflect the impact of the action on the scenario.

`PickupAtHouse` The bus can pick up a pupil if the pupil is at a house and the bus is in front of that house. We depict the rule for this in Figure 2.6a.

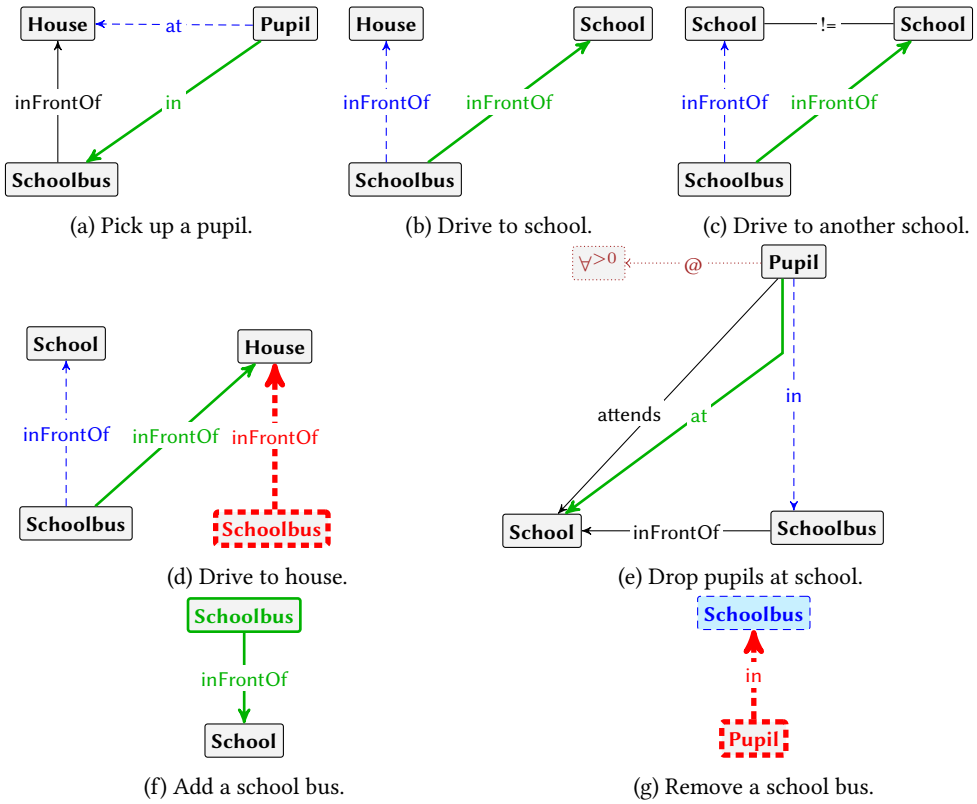
`DriveHtS/DriveStS` The bus can drive from a house to a school (Figure 2.6b) or from a school to another school (Figure 2.6c). Notice that a bus is not allowed to drive from a school to the same school again.

`DriveStH/DriveHtH` The bus can drive from a school to a house (Figure 2.6d), but due to limited parking space, this is only possible if there is no other bus in front of the house. The rule to drive from a house to another house is analogously defined.

`DropPupils` If the bus is at a school, it can unload all pupils in the bus which attend that school (Figure 2.6e) The ∀-node indicates that the rule should be applied to all matching pupils at once. Notice that we only consider this action, if there is at least one pupil in the bus which attends the school.

`BuyBus/SellBus` We can also buy new busses, which start in front of a school (Figure 2.6f), or sell (Figure 2.6g) them whenever no pupils are in the bus.

We illustrate a particular development of the initial scenario by the sequential application of the rewrite rules presented. For compactness, we choose not to depict identifiers or addresses, as they're not of importance here. The graphs are depicted in Figure 2.7 on page 23



(a) Pick up a pupil. (b) Drive to school. (c) Drive to another school.

(d) Drive to house. (e) Drop pupils at school.

(f) Add a school bus. (g) Remove a school bus.

Figure 2.6.: The different rules for corresponding actions.

1. The bus picks up the pupil at the house where the bus starts (apply `PickupAtHouse`).
2. The bus drives to another house (apply `DriveHtH`).
3. The bus picks up the pupil at the other house (apply `PickupAtHouse`).
4. The bus drives to the school (apply `DriveHtS`).
5. The bus drops off all pupils attending the school (apply `DropPupils`).

Of course, many other developments of the input graph are possible. ▲

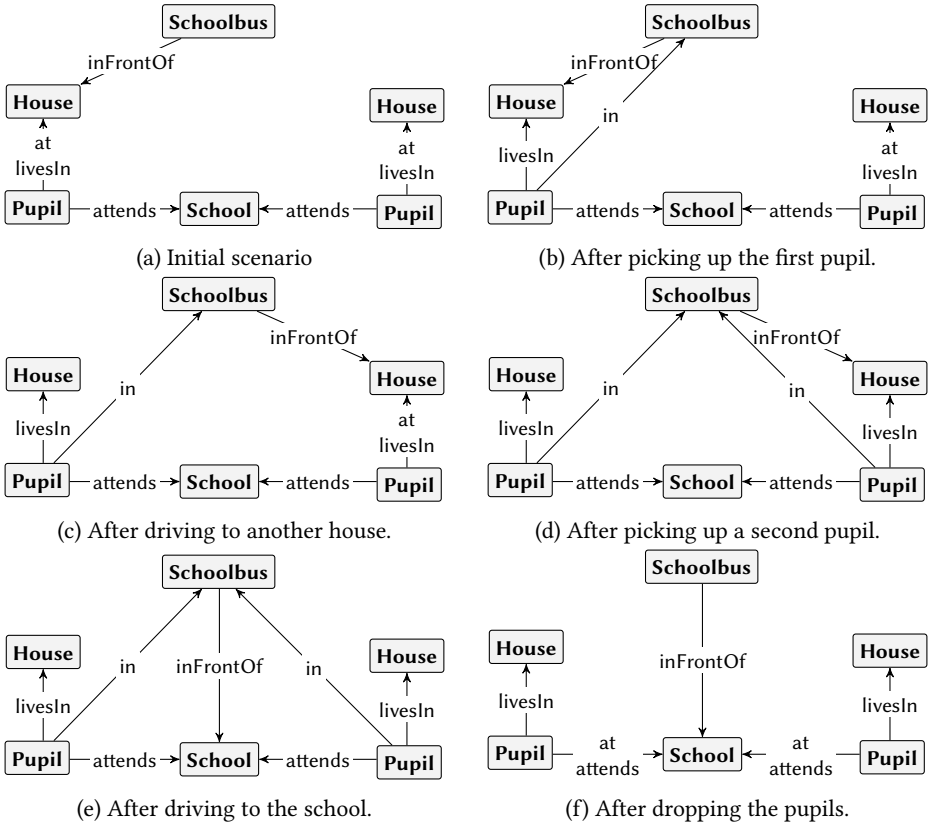
The example shows that often, multiple rules are applicable. Grooveconstructs states, which contain a particular instance of a graph, and connects these states via transitions labelled with a rule-name. A transition $s_i \xrightarrow{r} s_{i+i}$ denotes that the graph at s_{i+i} can be obtained by applying rule r on s_i. The states together with the transitions constitute a *labeled transition system* (LTS).

Example 2.5. We consider the setting from Example 2.4. In Figure 2.8, we depict part of the labeled transition system. An *open state* is a state which has other outgoing transitions, but that these are not explored. A *closed state* is a state for which all outgoing transitions are depicted. The path via the left (states s_0 to s_5) correspond to the scenario from Example 2.4 as depicted in Figure 2.7. The right path to s_5 depicts a path in which the bus takes the first pupil to school before picking up and dropping the other pupil. The triangles at s_1, s_2, s_6 and s_7, s_8, s_9 show pathes in which the bus potentially takes a detour via another building. ▲

2.3.2.3. Groove control programs

For various reasons, we might not want to construct the full labeled transition system. Relevant reasons are that

- the LTS might be infinite, or
- we are interested in reaching particular state and we have extra information how to find it, or
- some rules should be applied in a given order based on additional constraints.



(a) Initial scenario

(b) After picking up the first pupil.

(c) After driving to another house.

(d) After picking up a second pupil.

(e) After driving to the school.

(f) After dropping the pupils.

Figure 2.7.: One possible development of the initial scenario.

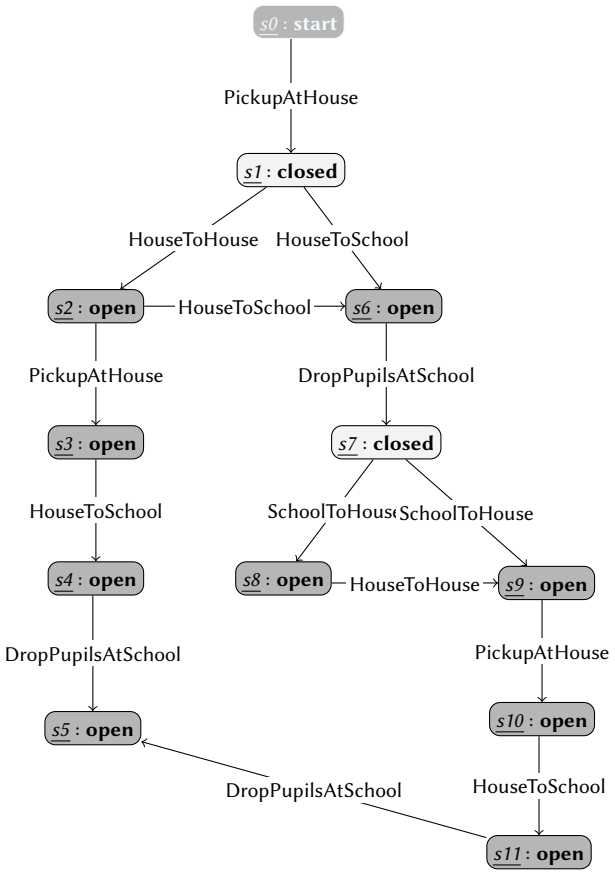


Figure 2.8.: Part of a labeled transition system.

To this end, Groove supports:

- *control programs*, which allow the user to restrict the possible applications of rules. A full specification of this language is not in the scope of this thesis, but given in [RBKS12]. In the next example, we discuss a simple control program.
- *strategy*, in which the way of exploration can be set, e.g. the user can enforce breadth-first search through the LTS.

Example 2.6. We continue with the scenario from Example 2.4 on page 21. We develop a control program (given in Listing 2.1) which prevents the bus from driving to a building without picking up or dropping pupils. We construct two *recipes* for this, each containg a *choice*. A choice is a splitting point in the program which says that either of the blocks separated by an or statement should be applied. A recipe is a method which is only applicable if there is a sequence of rules in the recipe which are all subsequently applicable. The `getPupil` recipe states that the bus should either

- first drive from a house to another house and then pick up a pupil at that house, or
- first drive from a school to a house and then pick up a pupil at that house.

The recipe ensures that at least one of the options above have to fully applicable, otherwise the recipe is not considered applicable. The control program contains a main part, which starts at line 11. The bus first picks up the pupil at the house where the bus starts, and then applies either `getPupil` or `dropPupil` *as long as possible* (alap). As long as possible means as long as either `getPupil` or `dropPupil` is applicable.

Listing 2.1: Control program

```
1  recipe getPupil() {
2      choice      {DriveHtH; PickupAtHouse;}
3      or          {DriveStH; PickupAtHouse;}
4  }
5
6  recipe dropPupil() {
7      choice      {DriveHtS; DropPupilsAtSchool;}
8      or          {DriveStS; DropPupilsAtSchool;}
9  }
10
11 // program starts here.
12 PickupAtHouse;
13 alap {
14     choice      getPupil();
15     or          dropPupil();
16 }
```

We depict the full LTS obtained by executing the control program in Figure 2.9. The continuous edges are now labelled with either a rule or a recipe. Dotted states are states created during applying a recipe. The *internal states* are states from which a next of the recipe can be applied, whereas *absent states* are states which are rejected, as no next rule in the recipe cannot be applied. The dotted transitions indicate which rule was applied. Moreover, we see that there is now a *result state*, which is a state where the control program has ended. The strategy influences the order in which the states are created.

If we want the bus to collect all pupils before driving to the school, we can change the main part of the program, as shown in Listing 2.2. Here, we always *try* to get another pupil, and only drop pupils at school if this fails. The LTS obtained by this control program contains only the left path depicted in Figure 2.9.

Listing 2.2: Alternative main program

```
12 PickupAtHouse;
13 alap {
14     try         getPupil();
15     else        dropPupil();
16 }
```

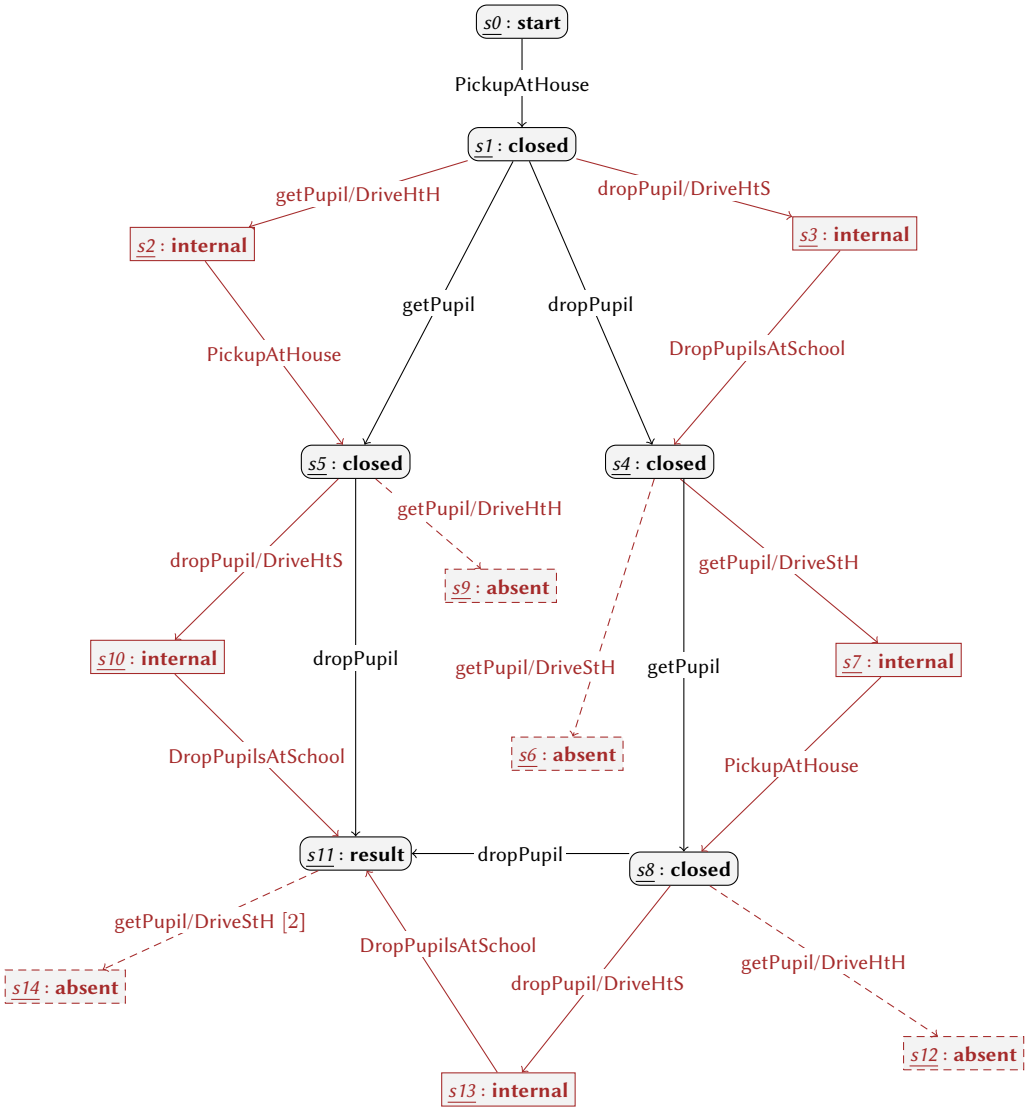


Figure 2.9.: Full LTS from control program.

▲

3. On Fault Trees

In this chapter, we discuss fault trees. After a brief introduction in fault tree analysis, we consider static fault trees (SFTs), describe how quantitative measures can be defined on them and discuss some of their deficiencies. We continue with the introduction of dynamic fault trees (DFTs) and show how the deficiencies of SFTs are overcome by DFTs. We discuss the underlying concepts (which we call *mechanisms*) and show how to define quantitative measures on DFTs. We then illustrate some of the semantic intricacies which are due to the informal definition of DFTs. Following those, we present a collection of DFTs found in the literature and give a brief summary of the existing DFTs. These case studies are used in later experiments. We conclude the chapter by an overview of existing semantics by giving their characteristics, as well as the restrictions they put upon DFTs, and showing some of the different choices regarding the earlier described intricacies.

3.1. Fault tree analysis

Fault tree analysis (FTA) is a method in reliability engineering. The concepts of *fault* and *failure* are elementary in the field. Several more or less equivalent definitions for the terminology exists, for an overview, see [Meu95]. We use the definitions from [ISO 24765].

> “ 3.1122 fault
>
> 1. a manifestation of an error in software.
> 2. an incorrect step, process, or data definition in a computer program.
> 3. a defect in a hardware device or component. *Syn: bug*
>
> NOTE: A fault, if encountered, may cause a failure. ”

> “ 3.1115 failure
>
> 1. termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. ISO/IEC 25000:2005 (...).
> 2. an event in which a system or system component does not perform a required function within specified limits.
>
> NOTE A failure may be produced when a fault is encountered. ”

We clarify the difference with an example. It is important to notice that the difference between fault and failure depends on the scope of our analysis.

Example 3.1. Consider a computer with two memory modules. When one of the memory modules has a defect, this is fault in the computer. If now both memory modules are defect, these faults lead to a computer failure. Please notice, when we would consider the memory module as system, the fault would indeed be a failure. ▲

Now fault tree analysis is described in [VS02] as an top-down approach to analyse the an undesired-event, a failure, in a given system. Then, this system is analysed

> “ (...) in the context of its environment and operation to find all realistic ways in which the undesired event (top event) can occur. The fault tree itself is a graphic model of the various parallel and sequential combinations of faults that will result in the occurrence of the predefined undesired event. The faults can be events that are associated with component hardware failures, human errors, software errors, or any other pertinent events which can lead to the undesired event. A fault tree thus depicts the logical interrelationships of basic events that lead to the undesired event, the top event of the fault tree. ”

Although fault trees are a qualitative model, i.e. they relate binary events in order to state whether a system failure occurs in the presence of given faults, they can be used for quantitative analysis whenever quantitative information about the occurrence of the faults is present.

As we focus on the analysis of fault trees, we use some simplified phrasing in the remainder. We use *the system failure* to refer to the failure which is described in the fault tree under some given context and environment. We talk about *the system behaviour* to refer to the "various parallel and sequential combinations of faults that will result in the occurrence" of the system failure. We use faults to refer to any input of a fault tree. We assume that the faults are statistically independent. We say that a component fails to refer to the occurrence of a fault. As long as a component has not failed, it is called operational.

Whereas fault trees are also an documentation effort, we focus on the quantitative analysis of fault trees. Therefore, elements of a fault tree which do not affect such analyses are not considered.

3.2. Static fault trees

Static fault trees are graphical modelling language to describe the combinations of faults that lead to the undesired *top-level event*. They were invented at Bell Laboratories for the analysis of missile systems [Eri99].

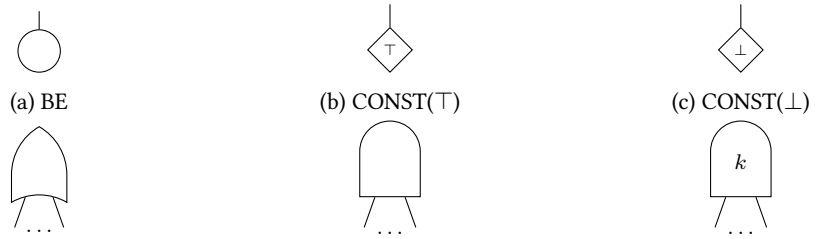
SFTs are directed acyclic graphs with a unique source (*top-level element*, top) and a set of sinks. The set of vertices which are reachable from a vertex v via one edge are called the *successors* of v.

Remark 6. A formal treatment of fault trees is presented in Chapter 4 on page 71.

The vertices of the graph are called *elements* and can have different *types*, which we introduce next. We then introduce quantitative analysis of the fault tree. This section ends with a discussion of the limited expressive power of static fault trees.

3.2.1. Static elements

Please notice that while the tree is constructed top-down, failures propagate from the bottom to the top, i.e. from the successors of a element to that element.



(a) BE (b) CONST($\top$) (c) CONST($\bot$)

(d) OR (e) AND (f) VOT(k)

Figure 3.1.: Element types of static fault trees.

We first introduce types for the sinks in the DAG.

- A *basic event* (BE), depicted in Figure 3.1a, represents a component of the system. Basic events do not have any inputs, i.e. are the sinks of the DAG. Their failure represents the failure of the represented component. Notice that we assume that the basic events are statistically independent, and therefore, each fault in the system is represented by at most one basic event. We often identify components by the basic event which represents them.

- A *constant element* is an element which either encodes a fault which is always present called *constant faults*(given-failure elements), or is *fail-safe*, i.e. it never fails. The two types are depicted in Figure 3.1b and Figure 3.1c, respectively. Constant elements are found with different names in the literature (often with distinct modelling use cases), as house event, undeveloped events (for fail-safe elements), or evidence (for constant faults).

All elements which are not sinks in the DAG are called gates. Gates propagate a failure of their successors depending on some type-specific condition over the successors. If the condition is fulfilled, we consider the gate failed.

Next, we present the three standard gate types for standard fault trees. We omit the inhibit gate, as, regarding quantitative analysis, it is equivalent to the and-gate. The possibility and consequences of adding a xor-gate or a not-gate are discussed in Section 3.2.3 on page 31.

- The *or-gate* (OR), depicted in Figure 3.1d, has failed if it at least one of its successors has failed. Its typical use is that a subsystem-fault can be caused by several different faults. It corresponds to the logical or operation.
- The *and-gate* (AND), depicted in Figure 3.1e, has failed if all of its successors have failed. Its typical use is that a subsystem-fault is caused by a combination other faults. It corresponds to the logical and operation.
- The *voting-gate* (VOT(k)), depicted in Figure 3.1f, has failed if at least k of its successors have failed, where k is a given *threshold*. The or-gate is a voting-gate with a threshold of one, the and-gate is a voting-gate with a threshold which is equal to the number of inputs. Moreover, the voting gate can be simulated by AND and OR.

Remark 7. We use rectangular boxes directly on top of elements to add identifiers to the elements.

The next example shows an SFT.

Example 3.2. Consider the SFT as shown in Figure 3.2. It depicts a fault tree for computer hardware. We construct the SFT in an hierarchical manner, based on the following description. The computer hardware (CH) is assumed to fail if the processor unit, or the memory unit, or the disk unit fails. The processor unit (PU) fails if either the processor or the fan fails. We do not further develop reasons for a processor (P) failure, but assume it is a basic event. The fan (F) is assumed to be fail-safe in the modelled scenario. The memory unit (MU) fails if both memory cards (M1, M2) have failed. Failure of memory cards is considered to be a basic event. The disk unit (DU) fails if two of the three disks fail. While the failures of the first two disks are considered basic events (D1, D2), we assume the third (D3) has already failed. ▲

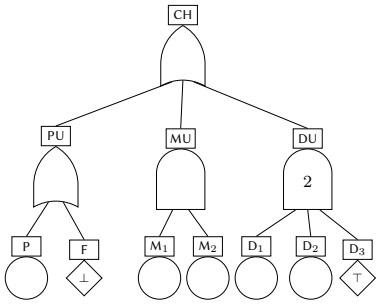


Figure 3.2.: A static fault tree for computer hardware.

3.2.2. Quantitative properties of a fault tree

A (static) fault tree of a system can be used as basis for a quantitative analysis, if appropriate information about the components is available. In this thesis, we assume that a continuous probability distribution describing the probability of failure between 0 and t, for some $t \in \mathbb{R}_{\geq 0}$. We will call such a distribution the *failure distribution of a component.*

Given a fault tree F for a system, the failure distributions for each component represented by a basic event, and a system life time T, we are mostly interested in two properties.

- *Reliability* $R_F(T)$

  Reliability, in general, is often defined similar to the following definition from [ISO 24765]

> “ 3.2467 reliability
>
> 1. the ability of a system or component to perform its required functions under stated conditions for a specified period of time.
>
> (...) ”

In the context of quantitative analysis, the following definition from [IEC60050-191] is particularly significant.

> “ 191-12-01 reliability
>
> the probability that an item can perform a required function under given conditions for a given time interval (t1, t2). ”

In the remainder, we use reliability always under the assumption that $t_1 = 0$. The *reliability function* maps a value for t_2 to the reliability for $(0, t_2)$.

The *reliability of a system* is the probability that no system failure occurs during the given system life time. As a concrete example, for a satellite which is expected to work for 10 years, the reliability gives the probability that the satellite does not fail within 10 years. In the context of FTA, the reliability becomes the probability that the top-level element of a fault tree does not fail.

The *reliability function of* F, denoted R_F, maps a time to the corresponding system reliability. It corresponds to $1 - \mathrm{FD}_F$, where FD_F is a continuous probability distribution describing the probability that the top-level element of F has failed before t. As a consequence, R_F is monotonically decreasing.

For SFTs, either $R_F(0) = 0$ (consider a constant failure) or $R_F(0) = 1$. For the latter case, we can further distinguish those cases where $\lim_{T\to\infty} R_F(T) = 0$ (consider a fail-safe element) and those with $\lim_{T\to\infty} R_F(T) = 1$. SFTs without constant elements all have $R_F(0) = 1, \lim_{T\to\infty} R_F(T) = 0$.

- *Mean Time To Failure* MTTF_F

  We give the definition from [IEC60050-191][1].

  > “ 191-12-07 mean time to failure
  >
  > The expectation of the total time duration of operating time of an item, from the instant it is first put in an up state, until failure, (...) ”

  Given the reliability function R_F, using the definition of expected value from Definition 2.4 on page 8, we derive

  $$\mathsf{MTTF}_F = \int_0^\infty 1 - R_F(t)dt$$

  We notice that for scenarios with $\lim_{T\to\infty} R_F(T) \neq 0$, MTTF_F may not exist, in which we define it to be infinity.

A thorough formalisation of these (and other) properties, as well as an overview over the existing algorithms is given by Ruijters and Stoelinga in [RS14]. Here, we only introduce *minimal cut sets* as this notion is helpful in later chapters.

A *cut set* for an SFT is a subset of the basic events in the fault tree such that the occurrence of faults for all these basic events causes a system failure. A minimal cut set is a subset of a cut set such that each proper subset of the minimal cut set is not a cut set. With the cut sets given, the system reliability is easily deduced from the failure distributions.

[1] We merged the definitions from mean time to failure and time to failure.

Example 3.3. The set of minimal cut sets for the SFT depicted in Figure 3.2 on page 29 is

$$\{\{P\}, \{M_1, M_2\}, \{D_1\}, \{D_2\}\}.$$ ▲

Assumption 2. *Although failure distributions can be arbitrary probability distributions, in this thesis, we assume the failure distributions for components to be exponential distributions, and use the* failure rate *to describe the failure distribution.*

Besides of referring to system reliability or MTTF, we also use *system performance* to refer to an arbitrary but fixed measure. An improved performance means a higher reliability and/or a higher MTTF[1].

3.2.3. Deficiencies of static fault trees

Static fault trees are both well-understood and heavily used. Their very limited expressive power has led to many extensions. We describe some systems where important characteristics cannot be modelled correctly by static fault trees.

Warm and cold spare components Many safety-critical systems feature redundancy in order to improve safety and dependability. We can distinguish three different types of redundancy.

1. hot redundancy, also *active standby.*
2. warm redundancy, also *passive standby.*
3. cold redundancy, also *cold standby.*

In the remainder, we often consider cold/hot standby a special case of passive standby. We illustrate the difference in the following example.

Example 3.4. We consider an electromechanical crossing barrier with two motors, with different system configurations. We assume a primary (first) motor to perform mechanical work in all cases. We consider the second motor to be redundant. The three cases below reflect the three types of redundancy.

1. Both motors are running and performing mechanical work.

   If the second motor is running and performing work, it is on active standby, and its failure rate is not likely to be reduced.

2. Both motors are running, but only the first drives the pole.

   If the second motor is running, but not performing work, it is on passive standby. Even though the stress on its parts is reduced, they may still wear off, so the assumption that it won't fail is not applicable. However, it is fair to assume that the failure rate is reduced.

3. By default, the primary motor is on duty. Only if it fails, the second motor is turned on.

   If the second motor is not used until the first motor has failed, it is on cold standby. It is a fair assumption that it won't fail as long as it is not used. ▲

With static fault trees, we cannot change the failure rate of a component based on the already failed components, which leads to *overestimated failure probabilities* involving the failures of components which are initially in passive standby.

Sharing of spare pools Many systems consist of spare components which are not dedicated to a particular task as long as they are unused. We notice that such spare components are often in passive standby. Below we give a typical example.

[1] It is possible to increase the system reliability while decreasing the MTTF and vice versa, therefore, system performance is only used as an abstract concept.

Example 3.5. Consider a car with 4 wheels attached and which is only operational with four operational wheels attached. The car includes a spare wheel in case one of the attached wheels fail, e.g. if it has a flat tire. Now, whenever the first wheel breaks, we can use the spare to replace the broken wheel. However, if now another wheel (including the now attached spare wheel) fails, we cannot use the spare wheel to also overcome this issue, so the car fails. ▲

Simple configurations as presented above can be handled by voting gates, if the components are hot redundant. However, different components may have different pools of available spare components.

Temporal conditions The effect of a component failure to the overall system is often dependent on the state of the system. The state of a system itself is often dependent not only on the set of component failures which have failed, but also the order in which the components have failed. One example are the shared spare pools above. However, many systems require some external machinery to reconfigure the system. A typical example is given below.

Example 3.6. Consider a crossing barrier like in Example 3.4. We extend the model by adding a switching unit, which is able to disconnect the first motor from the pole and connect the second one. As soon as the first motor fails, a working switching unit enables the system to use the second motor.

By this description, we can deduce that the system fails as soon as either both motors have failed, or the first motor and the switching unit have failed. We can refine this model by observing that the switching unit is of no further use to the system after it has switched the motors. Thus, if the first motor fails before the switch fails, the switch will have enabled the second motor and any future failure of the switch doesn't affect the system's operational state. Thus, we deduce that the system fails as soon as either both motors have failed, or the switch and then the first motor have failed (in that particular order). ▲

Moreover, as with cold redundancy, many failures in a system can only occur after the occurrence of another event.

Example 3.7. Consider a pump with an electric motor. After a leakage occurs, a short circuit failure can occur. Notice that this short circuit cannot occur before the leakage. ▲

Moreover, the occurrence of an event might prevent other failures.

Example 3.8. Consider a valve, which can be stuck open or stuck closed. However, as soon as it stuck open, it is impossible that it gets stuck closed afterwards. ▲

With static fault trees, we can neither restrict the ordering of events nor handle order-dependent failures.

Feedback loops Many systems feature some kind of feedback loops. To model this with a static fault tree, we have to apply an inconvenient trick. Consider the following example, inspired by [VS02].

Example 3.9. Consider a power supply unit (PSU) which has to be cooled by a thermal unit (TU). A failure of the TU causes, amongst others, the failure of the PSU. As the PSU powers the TU, a failure of the PSU causes a failure of the TU. Following [VS02], such a subsystem is modelled by splitting the faults of the TU and PSU into internal faults and faults caused by other system. In Figure 3.3, part of a fault tree involving a PSU and a TU are depicted. ▲

Repairs Fault trees can be used for assessing both system safety and dependability. In many contexts, the foremost interest is in the probability of some system failure, and the fact that components might be repaired is not important. This might include systems which are hardly repairable, like launched satellites, or system failures which pose a direct safety threat, like broken brakes in any mobile system.

However, often, the modeller's interest is in the availability of a system. Consider server farms, where a short system downtime is not tragic, but the overall downtime should be minimal. Such

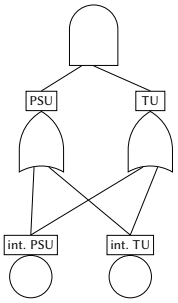


Figure 3.3.: Part of an SFT modelling a feedback loop.

systems are often repairable, e.g. by replacing failed components. Different strategies for repairing are proposed, ranging from components which restore their original state from their failed state with some rate or delay, yielding independent repair of components, to more complex strategies involving repair units which queue failed components and restore them, yielding repair strategies which depend also on the set of failed components.

In this thesis, we do not consider repairs in fault trees. For an overview, we refer to [RS14].

Non-coherent systems Static fault trees are *coherent*, which means that if some gate propagates failure at time t, it propagates failure also at $t + \delta t$. This is a limitation in two different senses.

- Some systems feature non-coherent behaviour, as is described by Chu and Apostolakis [CA80]. Contini *et al.* [CCR08] argue that the number of real-world non-coherent systems is rather small. Examples of non-coherent system are systems with special action in fail-dangerous states (i.e. system states which seemingly have a large probability to lead to a failure). As an example, consider precautions in chemical plants in case certain sensors do not work properly. In such systems, additional faults might lead a subsystem from fail-dangerous to actually failed. Such systems cannot be modelled correctly without non-coherent systems.
- Using non-coherent features within the fault tree may simplify the modelling, even if the overall system is coherent.

One common gate which features non-coherency is the xor-gate (exclusive or). As described in [VS02], this is often replaced by the (inclusive) or-gate. However, especially in the case of common cause failures, this may yield a very different result. In the case of static fault trees, the approximation is always conservative, i.e. the reliability and MTTF are under-approximated.
In this thesis, we do not consider non-coherent fault trees.

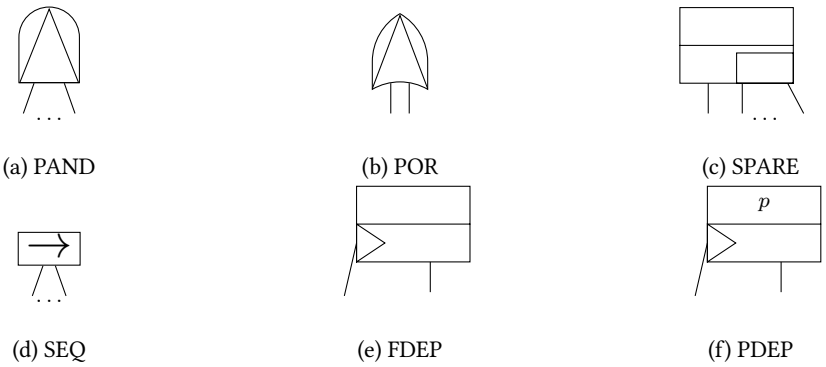
Repetition Often, many identical components and/or groups of components are present in the modelled system. Repeating subtrees leads to hard-to-read models and moreover, unfolding this repetition often leads to slower analyses in existing tool-support. In the literature, some effort is done to allow to use *replication*, e.g. by Bobbio *et al.* in [BFGP03].
In this thesis, we do not consider support for replication.

Multiple modes The concepts of different standby types, mutual exclusion and of repairs can be generalised to considering a component in different modes. As with standby types, the mode in which a component is may be dependent on the mode of other components.

We illustrate this with the following example.

Example 3.10. Consider the hot standby configuration of a crossing barrier, as described in Example 3.4 on page 31. The first motor, which is on duty by default, might be subject to additional stress after the second motor fails. Thus, the failure of the second motor might have an effect on the failure rate of the first motor. ▲

Remark 8. Furthermore, these modes can be an artificial construct to model non-constant failure rates, i.e. non-exponential failure distributions. Parallel and sequential combination of exponential distributions yields phase-type distributions, which can approximate any probability distribution, cf. [PH08].



(a) PAND (b) POR (c) SPARE

(d) SEQ (e) FDEP (f) PDEP

Figure 3.4.: Additional element types in dynamic fault trees.

3.3. Dynamic fault trees

We are now ready to examine a particular extension of static fault trees which feature some kind of internal state, which yields the attribute *dynamic fault trees*. Dynamic fault trees were first introduced by Dugan *et al.* in [DBB90].

We extend static fault trees by six other gate types. Moreover, we do not longer require the graph to have a unique source node. Therefore, we have to mark the top-level element. We do this by always labelling the element (with a rectangular box, as before) and adding an underscore to this label.

Furthermore, now we are adding elements which depend on the order of events, we highlight that the assumption of statistical independence implies that two component faults almost surely do not occur simultaneous. Therefore, we assume that component faults never occur simultaneous.

As DFTs support passive standby, each fault is extended with a passive failure rate, to a total of two failure rates. The passive failure rate may be 0 to reflect cold standby. Failure rates of 0 technically do not yield a valid failure distribution, however, this is merely a technical issue[1].

Whether basic events have a cold or warm standby component attached is not depicted, but this follows either from the adjoining description or the table with failure rates, if present.

3.3.1. Dynamic elements

Of the six additional element types described next, the priority-and gate, spare-gate, functional dependency and sequence enforcer are commonly included. The priority-or gate and the probabilistic dependency were introduced more recently.

Priority-and gate A *priority-and gate* (pand-gate, PAND) is an and gate which puts additional constraints on the order in which its successors fail. Usually it is agreed that the requirement is that the successors fail from left to right. The literature does not agree what happens in case two events fail simultaneous, we discuss this in Section 3.3.4.6 on page 48.

Pand-gates are commonly included in DFTs, and are also discussed as single extension to static fault trees in, among others, [YY08; XMTY+13] or in combination with other gates, e.g. in [WP10].

Example 3.11. We model the crossing barrier with additional switching unit from Example 3.6 on page 32. We depict the DFT in Figure 3.5. Starting from the top, we see that two conditions lead to the system failure SF. Either, via the left, the switch S fails before the motor M_A fails and then the motor fails, or, via the right, both motors M_A, M_B fail (in arbitrary order). ▲

Extending SFTs with priority-and gates makes the fail-safe element syntactic sugar as an infallible and-gate can be constructed by having two conflicting pand-gates as successors. In Figure 3.6a, using that two basic events do not fail simultaneously, we can be sure that P_1 (A before B) and P_2 (B before A) exclude each other (called: *conflicting*). Consequently, the condition P_1 and P_2 never holds, and the and-gate never fails. We would like to emphasise that already one pand-gate suffices for infallible subDFTs. In Figure 3.6b, again using that two basic events do not fail simultaneously,

[1] A formal treatment is included in Chapter 4 on page 71.

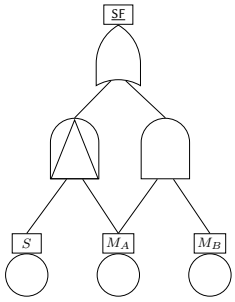
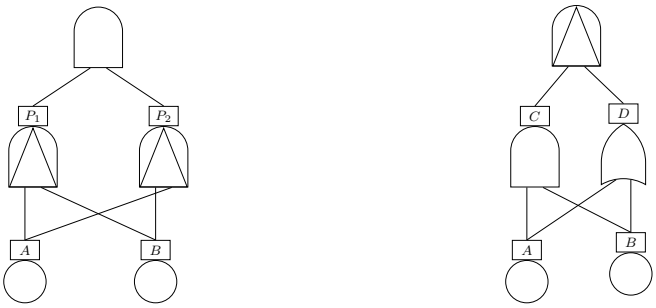


Figure 3.5.: Model for a crossing barrier with switch.



(a) Constructing fail-safe subtrees with conflicting pand-gates.

(b) Constructing fail-safe subtrees with a single pand-gate.

Figure 3.6.: Examples for constructing fail-safe subtrees.

.

either A or B fail before both A and B have failed, therefore, D always fails before C, thus the top never fails.

Priority-or gate The *priority-or gate* (por-gate, POR) is featured in Pandora (temporal fault trees[1]) by Walker *et al.* [WP09; WP10; EWG12]. We include the gate here for two reasons. First, it allows us to reflect the case study from e.g. [WP10] and [EWG12], and moreover, it allows a better review of some choices regarding priority-and gates in Section 3.3.4 on page 40.

The por-gate fails if the first successor fails before any of the other successors have failed. Notice that a (binary) por-gate is a kind of dual to the binary pand-gate, as a por-gate with successors A and B has failed if and only if a pand-gate with successors B and A is infallible.

Example 3.12. Consider two computing devices and an actuator connected via a data link. The system is considered operational as long as either of the devices is operational and no device blocks the data link, e.g. by turning into a "babbling idiot"[2]. Each device can have a processor failure, which causes no more activity of the device and therefore a failure of the device, or it can have a network link fault, which turns the device into a babbling idiot if the processor is still working. A babbling idiot leads to a direct system failure.

We depict the DFT in Figure 3.7. The top-level event fails if either both devices (D) fail, or an babbling idiot (BI) blocks the data line. Either of the devices fails if either (D_1) or (D_2) fails. Device i fails if either a processor (P_i) or the data link (L_i) fails. A babbling idiot blocks the link if either of the devices start babbling (BI_1, BI_2). Device i starts babbling if (L_i) fails and (P_i) has not occurred before. ▲

Spare gate *Spare gates* (SPARE) manage the usage of shared spare components and trigger the activation of components. Whenever a successor element representing the currently used *spare module* fails, the spare gate switches to the next available spare module, i.e. a successor element

[1] temporal fault trees ≈ static fault trees with priority gates

[2] Babbling idiots are devices which constantly send messages over a data link, thereby blocking communication of other devices.

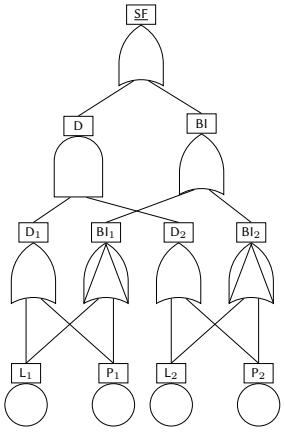


Figure 3.7.: DFT showing usage of POR gates.

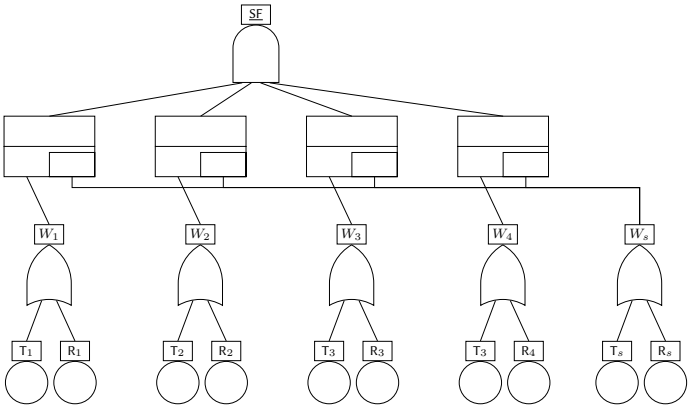


Figure 3.8.: The DFT for a car with a spare wheel.

which has not yet failed and is not used by another spare component. The next element refers to the ordering of successors from left to right. A spare module which is used is activated, that is, for all basic events in the spare module, the failure rate changes from the passive to the active failure rate.

Example 3.13. Consider a car and its wheels from Example 3.5 on page 32. Each wheel either breaks due to a broken rim or due to a flat tire. The car fails if one of its wheels fail and cannot be replaced. We depict the DFT in Figure 3.8.

As soon as the first tire or rim fails, the corresponding wheel W_i ($1 \leq i \leq 4$) fails. Now the spare gate above claims the spare wheel W_s, thereby activating the wheel - and thus its tire and its rim. Now, for a subsequent failure of another wheel W_j, $j \neq i$, its corresponding spare gate cannot claim the spare wheel anymore, and therefore, the spare and then the system fail.

Please notice that assuming a passive standby for the spare wheel adds a — less likely — scenario where the spare wheel fails before any of the primary wheels. In that case, nothing happens after the failure of the spare wheel, but any failure on the primary wheels immediately causes the system to fail. ▲

Originally, spare modules where limited to basic events, but these restrictions have been relaxed in recent work, e.g. in [BCS10]. Components belonging to a spare module S are in passive standby (with their passive failure rate) until S is activated. In the example above, everything connected to a successor of a spare gate (the tire and the rim in the example above) is part of the spare module. The following example shows the spare module of a more complex scenario.

Example 3.14. In Figure 3.9, we depicted a fragment of a DFT, with four spare modules, which we indicate by the dotted boxes. ▲

Indeed, a successor of a spare-gate represents a spare module. Successors of a gate v are in the

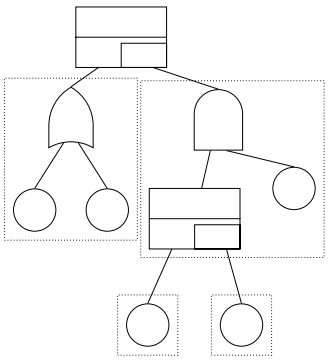
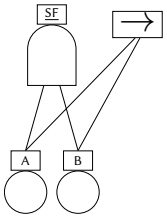
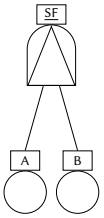


Figure 3.9.: Depicting spare modules.



(a) Restricting sequences via sequence enforcers.



(b) Regarding sequences via pand-gates.

Figure 3.10.: PAND vs. SEQ

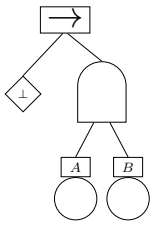


Figure 3.11.: Mutual exclusion with SEQ

same spare module as v, unless v is a spare gate. A more detailed discussion of the spare modules is given in Section 3.3.4.5 on page 46.

Sequence enforcer *Sequence enforcers* (SEQ) are used to guarantee a particular order of element failures. Please notice that sequence enforcers are unlike common gates, as they do not have an output, i.e. in the graph, nodes representing sequence enforcers have no incoming edges.

Example 3.15. Recall the water pump from Example 3.7 on page 32. The pump fails if a leakage (A) occurs and the motor has a short circuit (B). This short circuit can only occur after the leakage. In Figure 3.10a, we depict a DFT which correctly encodes the system. ▲

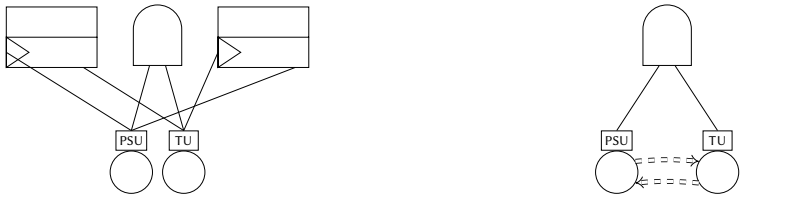
In [IEC60050], the sequence enforcer is only mentioned as an alternative to the pand-gate, which might be misleading. Whereas the pand-gate is a special and-gate which only fails if some order restriction is met, the sequence enforcer prevents certain orders from occurring. Consider the following example.

Example 3.16. Consider the scenario from the last example. In Figure 3.10a the pump fails after the failure of both A and B. The sequence enforcer ensures that the order in which A and B is accordingly, i.e. B does not fail before A. In Figure 3.10b we depict a DFT, which fails after the basic events A, B have failed in that particular order. The possibility that B fails before A does is not excluded, however, the DFT does not fail if first B and then A occurs. ▲

Sequence enforcers are very powerful gates. Among other things, they can be used to express mutual exclusion.

Example 3.17. Consider a binary sequence enforcer. The first successor has to fail before the second successor may fail. As a consequence, if the first successor never fails, then the second successor may never fail. In Figure 3.11, we depict a mutual exclusion of elements A and B, e.g. encoding a valve stuck open or stuck closed (cf. Example 3.8 on page 32). ▲

We discuss using cold spares to model sequence enforcers in Section 3.3.4.8 on page 50.



(a) Original notation (b) Alternative notation

Figure 3.12.: A DFT for a system with a feedback loop.

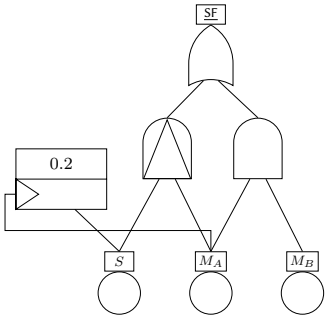


Figure 3.13.: The failure of the motor leads to a failing switch with probability 0.2

Functional dependency A *functional dependency* (FDEP) is an element type which forces the occurrence of basic event failures. As sequence enforcers, functional dependencies have no predecessors in the graph. The first successor of an FDEP is called *trigger*, all other successors are *dependent events*. If the trigger fails, then the dependent events should also fail. This introduces a second notion of basic event failure, namely *internal* or *forced* failures, alongside to the component faults which we also refer to as *external failures*.

Functional dependencies are helpful to model feedback loops. We consider the example given in [VS02].

Example 3.18. We consider the system with a feedback loop from Example 3.9 on page 32. We give the corresponding DFT in Figure 3.12. ▲

Besides the ease of modelling, functional dependencies are often used in combination with spare gates to work around the consequences of being a (indirect) successor of a spare gate. We discuss this in Section 3.3.4 on page 40.

Remark 9. To simplify depicted DFTs, instead of the functional dependency node, we often use a double dotted arrow to denote FDEPs. The origin of the arrow corresponds to the trigger, while the target corresponds to the dependent event. We depicted the identical DFT with the alternative notation in Figure 3.12b.

Probabilistic dependency The probabilistic dependency (PDEP) is an extension of the FDEP, and is included in the work of Montani *et al.* [MPBC06; MPBC08; MPBV+06]. It consists of a trigger input, as well as a non-empty set of dependent events, and contains a probability p.

As soon as the trigger of the PDEP fails, the dependent events fail, but only with probability p. Thus, a PDEP with $p = 1$ is equal to an FDEP. For $p = 0$, the dependent events are certainly not triggered and the PDEP is superfluous.

Example 3.19. Consider the switch from the barrier crossing (Example 3.6 on page 32). Besides an event with a continuous failure distribution, we could assume that as soon as the switch is actually used, it fails with a given (discrete) probability. We have modelled this in the DFT in Figure 3.13 ▲

3.3.2. Mechanisms in DFTs

When describing DFTs, it is useful to review the different mechanisms which are described in DFT elements.

- *Failure propagation*

  The foremost mechanism in a DFT is similar to failure propagation in an SFT. However, we see some differences. First of all, DFTs have two types of failure propagation. One is the usual propagation to the predecessor (hereafter: *failure combination*) in the graph, another is the propagation via functional dependencies (hereafter: *failure forwarding*). While failure combination is never cyclic, failure forwarding may (indirectly) cause the failure to propagate to the original element. Failure via functional dependencies also lets fail in two distinct ways, either via failure combination as some condition over their successors, or via failure forwarding.

  Moreover in the next section, we discuss how adding priority and spare gates introduces timing aspects and leads to questions about the temporal behaviour of failure combination and forwarding.

- *Module claiming*

  Spare gates require exclusive use of spare modules. This requires a mechanism to inform other spare gates that a subsystem cannot be used anymore. We call this mechanism claiming.

  Example 3.20. Recall the car with the spare wheel from Example 3.5 on page 32. The spare-gate for a wheel has to inform the other spare-gates as soon as it starts using the spare wheel, that is, after its currently (and initially) used wheel fails. ▲

  As soon as a used spare module fails, the spare-gate which used the module claims another module (represented by the successor of the spare-gate) which is not yet claimed. Then, the module is informed it is claimed. The module is then able to communicate to its predecessors that it has been claimed and is no longer available to these other spare-gates. We say that a spare-gate *uses* a module if the gate is active, it claimed the module, and it and has not claimed another spare module.

- *Activation propagation*

  To realise the support of reduced failure rates in case a component is standby, DFTs introduce an activation mechanism. Spare modules are initially considered inactive. Active spare-gates propagate the activation signal to the spare module they use. Inactive spare gates do not emit any activation signal to any of the spare modules. Thus, as soon as an already active spare-gate starts a successor, all BEs in the spare module are activated. It is important to notice that FDEPs do not propagate the activation signal.

- *Event prevention*

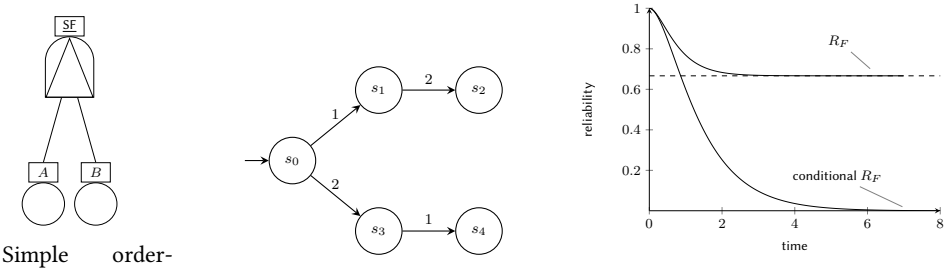
  With sequence enforcers, certain failure combinations can be explicitly excluded from the analysis. This is not limited to ruling out specific orders of basic event, but can also be applied to restrict certain claiming resolutions, although, in many cases, it requires ingenious fault trees.

  Moreover, cold standby components do not fail before activation, which leads also to a kind of event prevention.

3.3.3. Quantitative analysis of DFTs

DFTs are subject to the same quantitative measures as their static counterparts. As the ordering of the events influences the failure of the elements and the failure rate of subsequent events, the minimal cut set approach is not applicable to DFTs. Many different solutions have been proposed and are briefly sketched in Section 3.5 on page 67.

Besides the need for other algorithms, two additional measures for DFTs are useful. Whereas static fault trees either do never fail or almost surely fail at some point, for DFTs, it may be that after some sequences of all basic events, the fault tree has failed, and for others, the fault tree does not fail. The simplest such fault tree is a PAND with two basic events, as depicted in Figure 3.14a. For DFTs which have a non-zero chance of never failing, the MTTF as commonly defined may not exist. To adapt to this situation, we propose the following two measures.



(a) Simple order-dependent DFT. (b) The corresponding CTMC. (c) The reliability function plotted.

Figure 3.14.: Quantitative measures displayed.

- *Probability of Failure* $\Pr_F$.

  Describes the probability that eventually, the system fails. The following equality holds:

  $$\Pr_F = 1 - \lim_{t \to \infty} R_F(t).$$

- *Conditional MTTF* (CMTTF)

  Describes the expected time of failure under the assumption that the system eventually fails.

One eminent idea for defining algorithms is via a reduction of the DFT to a Markov chain. We use a trivial reduction in the following example to illustrate the different measures.

Example 3.21. Consider the DFT depicted in Figure 3.14a. We assume A to have a failure rate of 1 and B a failure rate of 2.

We build the CTMC depicted in Figure 3.14b as follows. Initially, we wait in s_0 for the failure of either A or B. After the occurrence of A, the CTMC is in state s_1. Only B can still fail, which is reflected by the single outgoing transition from s_1 to s_2. If we are in s_2, the top-level element has failed. If B fails first however, the CTMC goes to state s_3. A subsequent failure of A leads to s_4. The chance that the transition from s_0 to s_1 is taken equals $1/(1+2) = 1/3$.

In Figure 3.14c we have plotted the reliability function. Indeed, in the limit, it equals $1 - 1/3 = 2/3$. We also plotted the conditional reliability function, which is obtained by considering the subCTMC containing the states s_0, s_1, s_2. Based on the conditional reliability function, we obtain a CMTTF of 1.5, while the regular MTTF is infinity. ▲

Reductions to other stochastic models are discussed in Section 3.5 on page 67.

Another notion found in the literature (cf. [TD04; LXZL+07]) for SFTs with PANDs is the notion of *minimal cut sequences*, which is a generalisation of minimal cut sets. Indeed, it extends each cut set with ordering information. This method is not correct in general, as we discuss in Section 3.3.4.1.

3.3.4. Semantic intricacies of DFTs

In this section, we discuss a selection of potential pitfalls of DFT analysis. As the necessity of formal semantics has been stressed independently in, e.g. [CSD00; BCS07c], we do not repeat issues which are due to missing semantics. Instead, we discuss the possible choices in the semantics and the issues that arise if these choices are not carefully accounted for. A concise treatment of DFTs is key in the development of tools and algorithms for quantitative analysis, as otherwise unexpected results for DFTs are obtained. The presented intricacies are not independent of choices made in the semantics. However, great care was taken to present a broad range of possible choices for semantics. All intricacies originate from possibly undesired behaviour of, or claims made about, existing semantics.

An overview of the different intricacies w.r.t. fixed semantics is given in Section 3.5 on page 67.

3.3.4.1. Distribution of PANDs over ORs

This first issue is a common mistake which lies at the hart of issues arising with minimal cut sequences. It is commonly assumed that the two DFTs depicted in Figure 3.15 are equivalent,

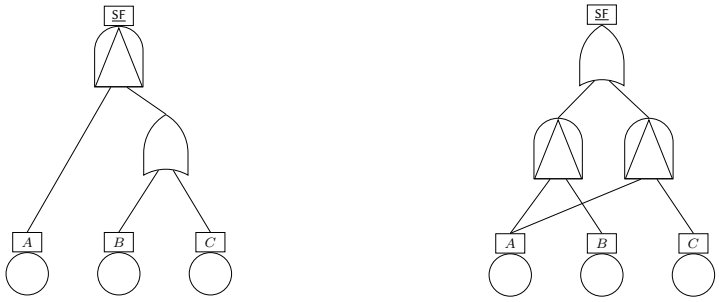


Figure 3.15.: Invalid distribution of or-gate over pand-gates.

which they are not. Consider the following order of failing basic events BAC. The DFT on the right-hand side fails, as A fails before C fails, and thus either of the two PANDs fail, causing the top-level element to fail. The DFT on the left-hand side does not fail, as the failure of B causes that the right child of the pand-gate fails before the left child.

Remark 10. In connection with minimal cut sequences, the cut sets for both DFTs in Figure 3.15 are given by $\{A, B\}$ and $\{A, C\}$. The cut sequences are then $\{AB, AC\}$. For the right-hand side, the information that C should not fail before AB and B not before AC is missing.

3.3.4.2. The necessity of a causal ordering

Many sources from the literature assume that both failure combination and forwarding is instantaneous. That is, as soon as a basic event fails, all predecessors whose fail condition is fulfilled fail, and all dependent events which are triggered fail. Both predecessors and dependent events may themselves cause failing elements in the fault tree.

This might lead to priority gates whose successors fail simultaneously, in which case it is often unclear what exactly is the semantics of a priority gate in case of simultaneous failures. However, this can be resolved by using a clear definition, e.g. here we assume that the priority gates fail if its inputs are triggered simultaneously.

A similar issue appears for spare gates. Multiple spare gates can fail at once, which is usually caused by the instantaneous failure propagation. The claiming of spare gates is then ill-defined, as an ordering is required for the claiming. We call this situation a *spare race*. Spare races can be resolved in numerous ways.

However, important in this context is the importance of respecting a causal order, as illustrated by the following example.

Example 3.22. Consider the following thought-experiment as depicted in Figure 3.16. Consider three spare-gates (A, B, C). The failure of P causes a spare race between A and B. Now assume that A wins the race and claims the spare modules S. Spare-gate B does not have any available successors left, and therefore fails. By failure combination D fails as well. The failure of D triggers the failure of Z, which means that C enters the spare race. With the assumption that claiming and failure propagation are both instantaneous, which means that C is racing at the same point to get a spare component. C however cannot win, as we used that A claimed it in the argument before.

We now consider some backward reasoning, showing that the failure of P and assuming that C claims S does not invalidate any property. If C claims S, then certainly Z must have failed. As we consider the initial failure of P, the only reason why Z fails is that D failed. D only fails if B failed. B fails if Y fails – B cannot claim S as it is claimed by C. Y indeed fails, as P causes Y to fail. ▲

The example given shows the need for some causal ordering. Notice that three mechanisms played together in the given example:

- Failure combination ($X \to A$, $Y \to B$, $Z \to C$ and $B \to D$).
- Failure forwarding ($P \to \{X, Y\}$, $D \to Z$).
- Successful claiming ($A \to S$) and unsuccessful claiming ($B \to S$).

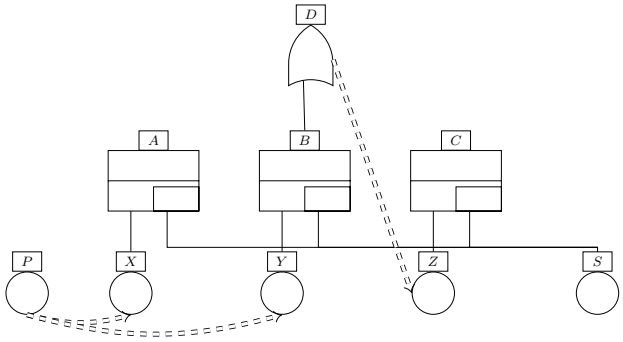


Figure 3.16.: Part of a DFT to illustrate missing causality.

Introducing a notion of causality for one of these mechanisms indeed resolves the issue described above under the assumption that common syntactical restrictions regarding the spare modules are applied. We discuss the syntactical restrictions on spare modules in Section 3.3.4.5 on page 46. A full discussion on which mechanisms should include a causal ordering is out of scope of this thesis. In the next paragraph, we briefly discuss the most-used method for implementing a causal order.

Causal and temporal ordering We observe that in DFTs, a notion of ordering is already embedded, e.g. in the definition of priority gates. This is usually interpreted as a temporal ordering, which leads to two different notions of ordering in DFTs.

However, one implementation of causal ordering is by assuming (infinitesimal) time steps for applying cause-effect relations. Thereby, only one order relation is present in the DFT. Moreover, this resembles the real behaviour of the modelled system, at least when applied to claiming or failure forwarding.

Example 3.23. Recall the feedback loop between a thermal and a power unit, as discussed in Example 3.18 on page 38. The failure of, e.g. the power unit triggers the failure of the thermal unit. Indeed, we could rephrase this and write that the power unit causes the subsequent failure of the thermal unit. We furthermore recall the car with a spare wheel from Example 3.5 on page 32. Indeed, we could describe the mechanism by writing that the claiming of the spare wheel by any of the gates causes the spare wheel to be unavailable for any subsequent replacements. However, if we inspect the failure combination in Example 3.6 on page 32, the failure of both motors does not happen after the last motor has failed, but exactly with the failure of the last motor. ▲

Remark 11. The distinct applications of failure combination and failure forwarding often go along the lines sketched in the previous example. However, strictly splitting these two types of failure propagation in DFTs is troublesome, as FDEPs are also used because they are typically able to transfer failures from one spare module to another, without being subject to exclusive claiming or activation, cf. the examples in Section 3.3.4.5 on page 46.

We furthermore observe that a causal ordering is a partial ordering. The standard interpretation of basic events places them in a total order, therefore, the standard interpretation of the temporal gates also uses this total order. If the causal order is resolved by a temporal argument, the semantics of a temporal gate are affected by the resolution of the causal order.

Example 3.24. Consider the DFT depicted in Figure 3.17. Assuming a causal ordering for failure forwarding, the following situation arises. An initial occurrence of C causes subsequent failures of A and B. Depending on the order of A and B, T fails. If we assume here that the causal order is resolved by a (total) temporal order, than A and B fail in some order, but not simultaneously. This causes a non-deterministic failure-behaviour of T, due to the (common cause, thereby simultaneous) failure of C. ▲

Thus, combining causal and temporal ordering leads to new questions. As an example, we may ask, which events can occur simultaneously. It could be argued that the infinitesimal time steps are almost surely not identical and that thus, failures of events due to causal dependencies do not occur simultaneous. But arguing the other way around is also possible, i.e. by stating that the forwarding

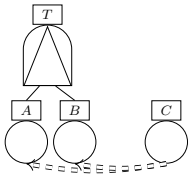
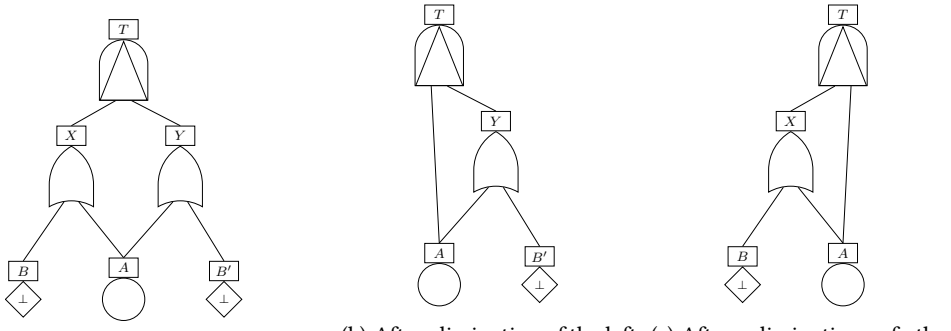


Figure 3.17.: Causal order and temporal ordering combined.



(a) The reference DFT.

(b) After elimination of the left or-gate.

(c) After elimination of the right or-gate.

Figure 3.18.: Illustrating the effect of ordered failure combination.

is not chaotic and might model deterministic behaviour whose timing is well-defined, and that any assumption ruling out such cases ignores potential interesting corner cases.

Moreover, especially the combination of the — often unexpected — causal relation due to failure combination and the temporal conditions of the gates leads to DFTs that are hard to analyse. In particular, gates do not only fail upon their failure condition being fulfilled, but they also influence the possible interleavings of gate-failures. In the next example, failure combination is assumed to be totally ordered. Put it differently, no two gates fail simultaneously. If this ordering is assumed to be a temporal ordering, seemingly equivalent DFTs have different interpretations.

Example 3.25. Consider the DFT depicted in Figure 3.18a. Please notice that X fails if and only if A fails, as A is the only failable successor. Therefore, we could expect that changing the first successor of T to A does not change the semantics. The resulting DFT is depicted in Figure 3.18b. Analogously, the elimination of Y yields the DFT depicted in Figure 3.18c, and by an analogous argument, we presume it is equivalent.

We reconsider the reference DFT (Figure 3.18a). Under the assumption that the failure combination is totally ordered and that we use a temporal ordering for this, after a failure of A, either X fails before Y or Y before X. Whether the pand-gate T fails is depending on the used ordering. Now, the two "simplified" DFTs differ fundamentally. In Figure 3.18b, the first successor of T (A) surely fails before the second successor (Y). This leads to a failure of T. In Figure 3.18c, the second successor of T (A) surely fails before the first successor of T (X). Therefore, T does not fail. ▲

The embedding in claiming yields also room for different interpretations, which we do not discuss here. To summarise, while merging a causal ordering with a temporal ordering seems natural and appropriate in many situations, it leads to delicate issues when interpreting DFTs. In particular, it seems more natural to assume true concurrency semantics for failure combination.

3.3.4.3. Claiming and non-determinism

In the last section, we saw that we might introduce non-determinism to handle the embedding of a causal order in a temporal order. Moreover, spare races might be resolved as being non-deterministic. Some authors have argued against non-determinism in fault trees, e.g. Merle *et al.* in [MRLB10] argue that especially critical infrastructures should be designed as deterministic.

On the other hand, especially in systems where human action is involved, policies might not be as precise (or precisely followed) that we could consider the system deterministic. Moreover,

FTA is not only applicable to existing systems, but might also be applied during design time, as described in [VS02]:

> “6. Use of FTA to assist in designing a system. When designing a system, FTA can be used to evaluate design alternatives and to establish performance-based design requirements. In using FTA to establish design requirements, performance requirements are defined and the FTA is used to determine the design alternatives that satisfy the performance requirements. Even though system specific data are not available, generic or heritage data can be used to bracket performance. This use of FTA is often overlooked, but is important enough to be discussed further (...)” ”

During design-time, the lack of specific data might be handled by non-determinism due to underspecification. Nevertheless, we agree that it is valuable to have support for deterministic claiming policies in DFTs.

Non-deterministic models may be resolved by assuming a probability distribution over the possible resolutions of the non-determinism. In [CSD00], a uniform distribution is assumed. This choice is accounted for by the assumption that the spare-modules have equal properties. This assumption, however, is not enforced by the semantics. Moreover, in [CSD00], spare modules are always basic events.

We give an example where the non-determinism can be resolved by any distribution - and therefore by the uniform distribution - as the outcome for each resolution is equivalent. We then show that many minor changes to the DFT cause systems in which the way of resolving spare races affects the behaviour of the DFT.

Example 3.26. Consider the DFT given in Figure 3.19a. The DFT describes a hypothetical communication system consisting of two radios R_1 and R_2, which both have to be operational. Each radio consists of an antenna (A_1 and A_2, respectively) and a power unit (P_1 and P_2, respectively). Both power units have their own power adaptor (PA_1 and PA_2, respectively). The power adaptors are connected to a common power supply (PS). Moreover, each power unit can use one of the two spare batteries (B_1, B_2). We assume that the failure distributions of B_1 and B_2 are identical.

In case PS fails, some kind of spare race is triggered. However, as there are spares for both P_1 and P_2, the actual order of claiming has no influence on the system. Even if there exists only one (available) spare module, each ordering of the spare race yields equivalent outcomes, as the second power unit which tries to claim a battery would fail, and therefore, the whole system would fail.

However, we could also assume that the system only fails if both radios have failed, i.e., if SF is an and-gate (or a spare gate, as depicted in Figure 3.19b), instead of an or-gate. In that case, having only one spare module potentially leads to different outcomes, either because one of the antennas might have failed before the spare race, or due to different failure distributions for the two antennas. Please notice, that with two available spare modules - and without the assumption that the failure distributions of the spare modules are identical, the outcome might also be dependent of the way the spare race is resolved. ▲

Besides the given example, if we consider subtrees as spare modules, then even isomorphic spare modules might behave differently upon claiming, given that some events of the module already failed before, without causing the failure of the whole spare module. Such modules fail potentially sooner after claiming than their fully operational counterparts.

Non-deterministic claiming allows to construct non-deterministic failure propagation, which can than be used arbitrarily throughout the DFT.

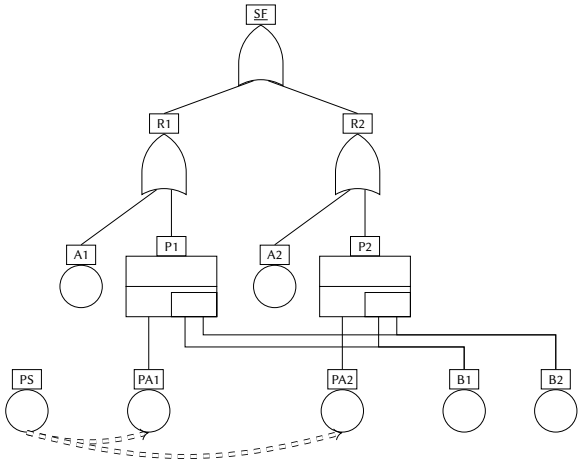
3.3.4.4. Early and late claiming

Conceptually, there are two important differences between claiming and activation.

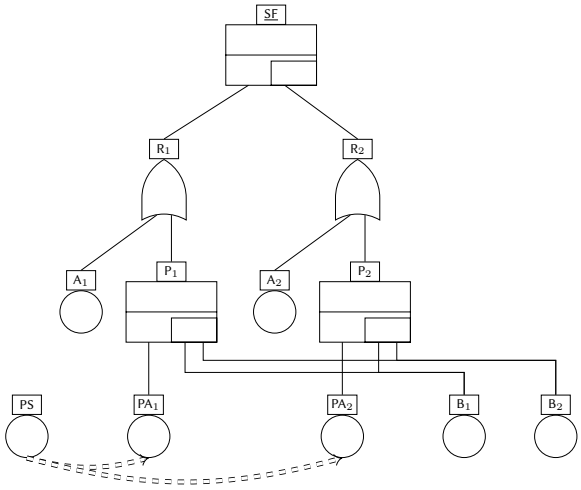
1. The moment a module is being used somewhere is not necessarily the moment it is activated.
2. Components which are not subject to exclusive claiming might also be inactive or active.

In this section, we discuss the former. The latter is described in the next section.

One may think of numerous scenarios in which spare modules get assigned to a dedicated task without actually performing the task, e.g. scenarios in which hardware is manually replaced. If we



(a) Both radios required for operation.



(b) The second radio is redundant.

Figure 3.19.: DFT of a communication system.

limit this discussion to existing DFT behaviour, then this distinction is visible in DFTs with nested spare gates, i.e. spare modules which contain spare-gates. In particular, we can think about three different claiming behaviours in those cases where a used spare module of an inactive spare gate fails:

Early claiming. The claiming behaviour of a spare gate is independent from its activation status. It is ensured that an operational spare gate has claimed an operational successor.

Late claiming with late failing. Inactive spare gates do not claim a successor. As soon as a spare gate is activated, it is ensured that the gate has claimed a subsystem, otherwise it fails upon activation. Inactive spare gates never fail.

Late claiming with early failing. Inactive spare gates do not claim a successor. It is ensured that an inactive and operational spare gate has an available successor, otherwise it fails with the last successor becoming unavailable.

We illustrate the difference in the next example.

Example 3.27. Consider the DFT depicted in Figure 3.19b, originating from a communication system as described in Example 3.26. We assume here that the second radio R_2 is in passive standby.

Consider the failure of just PA_2. If we assume early claiming, then the power unit of the second radio directly claims some batteries, which are then not available any more for P_1. If, on the contrary, we assume late claiming, then P_2 does not claim any of the batteries. Only after the

failure of R_1 and the subsequent activation of R_2, P_2 tries to claim a battery. With early failing, P_2 fails if either both B_1 and B_2 have failed or either of them have failed and the other is claimed by P_1. With late failing, P_2 fails only if it fails to claim something upon activation of R_2. ▲

Which behaviour fits best depends on the use case and cannot be fixed in general. We notice that both behaviours have some semantical consequences which go beyond the described situation.

On one hand, late claiming introduces failure due to claiming or activation, respectively. While with early claiming, a spare-gate is only claiming after a successor has failed, and therefore, it does not fail to claim without a successor failing at the same moment. With late claiming with early failure, spare gates may fail due to successors being claimed by other spare gates. Moreover, the claiming may then cause spare races, which results in event propagation and claiming becoming interdependent. Analogously, for late claiming with late failure, spare gates may fail upon activation. Thereby, activation may cause spare races, and event propagation and activation become interdependent.

The next example shows a scenario for failure upon claiming and failure upon activation.

Example 3.28. We continue with Example 3.27. We consider the subsequent occurrence of PS B_1 B_2.

First, we assume late claiming with early failing. P_2 does not claim any battery as it is not yet active. P_1 thus claims B_1. After the failure of B_1, P_1 claims B_2. Now, no successor of P_2 is available anymore and it has not claimed anything, thus it fails.

Second, we assume late claiming with late failing. As above, P_1 claims B_1 and P_2 does not claim anything. After PS, After the failure of B_1, P_1 claims B_2. The failure of B_2 causes the failure of P_1, and R_2 is activated. Now P_2 is activated. As it can not claim any successor, P_2 fails. ▲

On the other hand, using late claiming allows uniform treatment of the two mechanisms, which reduces the complexity of the behaviour especially in DFTs without nested spares, as a spare module is active if and only if it is claimed. Early and late claiming are generally incomparable w.r.t. the common quantitative measures on DFTs.

3.3.4.5. Spare modules revisited

In this section, we continue discussing differences between claiming and activation. We focus on the precise extent of spare modules and the consequences thereof w.r.t. claiming and activation.

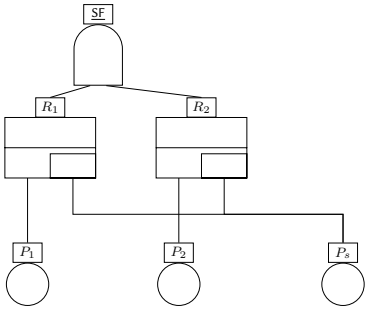
We start by restricting spare modules to basic events, as this is the original and most widespread variant. Activation and claiming coincide here. Thus, to activate a component, it has to be claimed. On the other hand is claiming exclusive, which leads to work-arounds to model system behaviour with DFTs. These work-arounds are not good practice as they are hard to understand.

Example 3.29. We consider a system consisting of two radios, which fail as soon as their power supply fails. The power supply can be replaced by a power generator, which is powerful enough to drive both radios. The failure rate of the power generator switches rises as soon as the generator is used, i.e. as soon as either of the primary power supplies fail. In Figure 3.20 we depict different DFTs for the system.

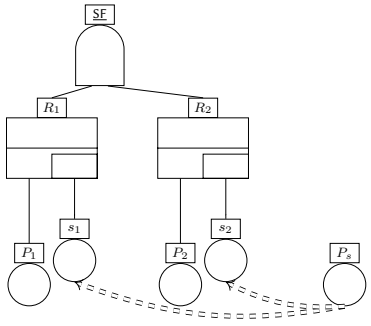
In Figure 3.20a, as soon as the primary power supply fails (P_1 and P_2, respectively), the corresponding radio (R_1, R_2) claims the power generator P_s. If the primary supply of the other unit fails, the DFT assumes the system fails, as the usage of the power generator is exclusive. Thus, the depicted DFT fails to model that the power generator is able to power both radios.

In Figure 3.20b, as soon as one of the primary power supplies fails, the radio claims their "connection" to the power generator (s_1 or s_2). However, only the connection is activated. The power generator is stays thus either active or passive, independent of the failure of the primary power supplies.

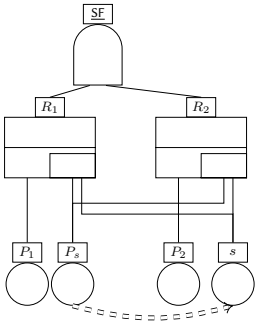
In Figure 3.20c, we give a work-around. The first unit with a failed power supply claims and activates the power generator. If the power supply of the other unit fails as well, it uses the connection, which fails with the power generator. Depending on the exact failure propagation behaviour, a failure of the power generator before the second primary power supply fails might cause the power unit which used the power generator before to claim the connection (s). However, this connection would then directly fail. ▲



(a) The power generator can only be used by one radio.



(b) The power generator never switches from passive to active.



(c) The DFT is using a workaround.

Figure 3.20.: Different attempts of modelling a shared power generator.

A straightforward solution to overcome the restrictions of exclusive claiming would be given by propagating activation in reverse direction through FDEPs. Upon activation of the dependent event, the trigger of such FDEP is then activated as well. However, there are also scenarios where such a reverse through-put is not wanted, cf. Example 3.18 on page 38.

If we allow more general spare modules, then we also see scenarios in which a module is activated under given circumstances, but does not add to the failure of the spare module with which it is activated. This is illustrated by the following example.

Example 3.30. Consider a spare pump in an industrial environment where a failing ventilator (vnt) of the pump doesn't affect the pump itself, but leads to failures elsewhere in the system. The ventilator is activated together with the pump. The DFT is depicted in Figure 3.21. ▲

Up until now, we have been rather imprecise about the precise interpretation of spare modules. We recall the accurate description depicted in Figure 3.9 on page 37. There, the spare modules were independent, i.e. unconnected, trees. In this section, we saw that this restriction leads to work-

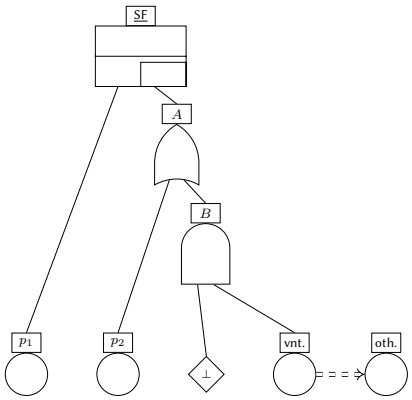


Figure 3.21.: DFT with an element whose failure does not contribute to its module.

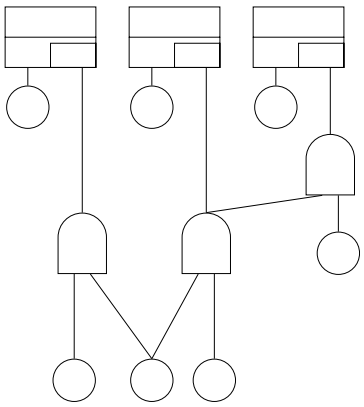


Figure 3.22.: Dependent spare modules.

arounds which do not agree with the hierarchical way DFTs should be created. Spare modules can be generalised from spare module trees to independent acyclic graphs, which simplifies modelling, cf. Figure 3.21. However, dropping independence yields multiple open questions. We review the DFT in Figure 3.22. There, spare modules are no longer represented by a unique represting gate. It remains open what happens if one of the primary spare modules fail. We do not cover further details in this thesis, as any extension towards this requires dedicated treatment and is not present in any existing semantics.

3.3.4.6. Selection of priority gates

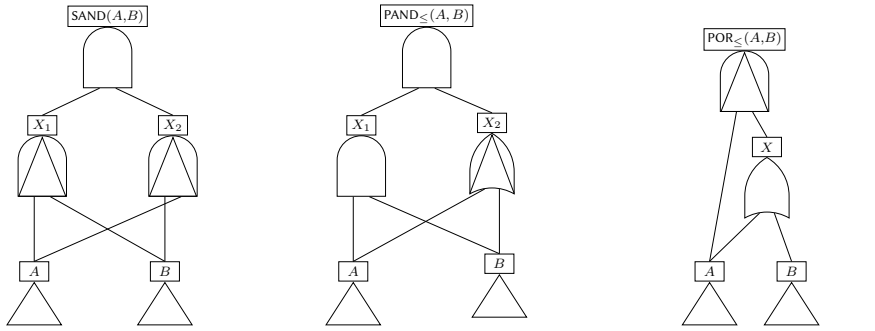
In standard DFTs, as described e.g. in [VS02] or [IEC60050], only priority-and gates are described. However, these sources do not clarify the proposed behaviour in case two inputs fail simultaneously. Both *inclusive* and *exclusive* pand-gates are found in the literature, see Table 3.2 on page 70. Furthermore, priority-or gates, though not as common as priority-and gates, simplify the modelling of specific situations.

In the following, we also consider the *simultaneous-and gate*, which has less practical relevance and is not a priority-gate (it is commutative) but eases the subsequent argumentation. Moreover, we consider the standard OR and AND. We start with depicting the behaviour of the gates in Table 3.1. Here, we describe the situation of two inputs, A and B, failing. Three scenarios are possible: A occurs before B, A occurs at the same moment as B, or A occurs after B. These three situations are listed in the table as $t(A) < t(B)$, $t(A) = t(B)$ and $t(A) > t(B)$, respectively. For each gate $G(A, B)$, the shaded area depicts the situations in which $G(A,B)$ is considered failed.

	$t(A) < t(B)$	$t(A) = t(B)$	$t(A) > t(B)$
	A B	A,B	B A
AND(A, B)			
OR(A, B)			
SAND(A,B)			
PAND$_{\leq}$(A, B)			
PAND$_{<}$(A, B)			
POR$_{\leq}$(A, B)			
POR$_{<}$(A, B)			

Table 3.1.: Behaviour of several binary gates with given occurrence of their inputs.

In the following, we assume instantaneous failure propagation, and that we are allowed to have AND and ORs. We observe the following.



(a) SAND from PAND$_{\leq}$. (b) PAND$_{\leq}$ from POR$_{\leq}$. (c) POR$_{\leq}$ from PAND$_{\leq}$.

Figure 3.23.: Using inclusive priority gates to simulate other inclusive gates.

If we add one of the inclusive priority gates, then the other inclusive gate as well as the sand-gate are syntactic sugar.

We show how the gates can be emulated by a series of other gates. In Figure 3.23a we use that requiring $t(A) \leq t(B)$ and $t(B) \leq t(A)$ is equivalent to $t(A) = t(B)$. In Figure 3.23b, we restrict the failure propagation of the POR by the requirement that both A and B have failed. In Figure 3.23c we route the input of A to both inputs of B. Please notice that only the last construction does not work for non-instantaneous failure propagation.

With the inclusive priority gates, we cannot simulate exclusive priority gates.

This statement can be deduced by noticing that all available gates fail whenever both inputs fail simultaneous. By an inductive argument, it is impossible to construct a subtree with two inputs whose top-level element does not fail if the inputs fail simultaneous.

With the exclusive POR, we can simulate all described priority gates.

Constructing an exclusive PAND with an exclusive POR is done analogously to the inclusive case (cf. Figure 3.23b). Assuming we can use a sand-gate, an inclusive POR can be constructed as depicted in Figure 3.24a. It remains to show that we can indeed construct a sand gate. Using exclusive PORs, we depict a possible construction in Figure 3.24b. We can use (exclusive) pand-gates as they can be simulated. To see why this construction is correct, let us first assume that A and B occur simultaneously. As the PANDs Z_1, Z_2 are exclusive, they do not fail, therefore, Y does not fail. On the other hand, X fails, and therefore, the top level exclusive POR has a left child which fails strictly before the right child fails. Now, if only A or B fails, then no further gates fail. If first A fails and B afterwards, then Z_1 fails, and therefore Y fails. Although X does also fail, the exclusive nature of the top level POR gate prevents the failure propagation. The remaining case where first B and then A fails is analogous.

Please notice that we can get from exclusive to inclusive gates because the standard AND and OR are, in some sense, inclusive gates as well.

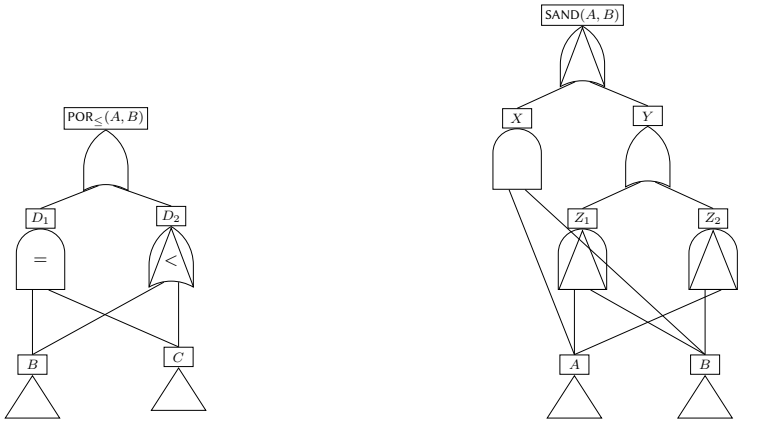
With the exclusive PAND, we cannot simulate any other of the described priority gates.

To substantiate this, we show the following two statements independently.

- *With the exclusive PAND together with AND and OR, we cannot simulate the exclusive POR.*

  The por-gate fails already after the failure of just A. Furthermore, we observe that no combination of just AND and OR suffices to model the POR. Thus, any solution uses at least one pand-gate. Such a PAND fails only if both successors have failed, and this was not simultaneous. Let t_1 denote the moment the first successor of such a PAND failed and t_2 the moment the second successor failed. The PAND has to fail after just the occurrence of A, thus A fails at t_2, and $t_1 \neq t_2$. However, then there has to be an event which failed at t_1, however, we considered a scenario with just A failing, and A fails not at t_1.

- *With the exclusive PAND together with AND and OR, we cannot simulate any inclusive priority gate.*



(a) POR$_{\leq}$ from POR$_{<}$ and SAND. (b) SAND from exclusive gates.

Figure 3.24.: Using the exclusive POR to simulate inclusive gates.

Using the statements above, it suffices to show that we cannot simulate the inclusive PAND. As above, any simulation requires the usage of the exclusive PAND. The exclusive PAND only fails after the failure of the non-simultaneous failure of its two successors. Now, if we consider the simultaneous failure of A and B, then we have only one moment of failure, so none of the exclusive PAND gates fail. Due to coherency, we can thus assume that no exclusive PANDs are present in the candidate subDFT, and that it is therefore no simulation of the inclusive PAND.

If we assume that the failure combination is not instantaneous, the used constructions are not necessarily valid, cf. Section 3.3.4.2 on page 41. With instantaneous failure combination, we get the full expressive power only when using the exclusive POR-gate, as it cannot be simulated by any other combination of gates, and the exclusive POR moreover suffices to simulate the other gates (in combination with AND and OR).

3.3.4.7. Under-approximation of exclusive-or

We recall from Section 3.2.3 on page 31 that xor-gates induce non-coherent behaviour, which is also not present in DFTs. For static fault trees, this is often circumvented by assuming or-gates for xor-gates. Whereas this approach is guaranteed to under-approximate the system performance for SFTs, this approach is not suitable for DFTs, as shown in the following example.

Example 3.31. Consider the DFT depicted in Figure 3.25. We assume the or-gate X is used to model an x-or gate. We consider the sequence BCD. Whereas the real system (with an exclusive or behaviour) fails after the additional occurrence of A and then E, the DFT cannot fail anymore as X is considered failed, thus Y and Z and thereby the ordering requirement of the PAND is violated. In this particular example, using an or-gate to model xor-behaviour over-approximates the system performance.

Please, notice that the precise behaviour of the pand-gate in case one of the successors is non-coherent is unspecified. Therefore, we use a somewhat verbose DFT in which we circumvent any unspecified behaviour. ▲

The difference is caused by the presence of elements such as PAND, where failed children may prevent the failure propagation of their parents. As SFTs are also DFTs, using or-gates for xor is neither guaranteed to be under- nor over-approximating the system performance.

3.3.4.8. Emulating sequence enforcers

Both [BCS10] and [MPBV+06] do not include sequence enforcers, as they claim that the induced behaviour of a sequence enforcer, supposedly, can be modelled by cold spares. We claim that only a small, yet interesting, fragment can be modelled. First, we give a positive example.

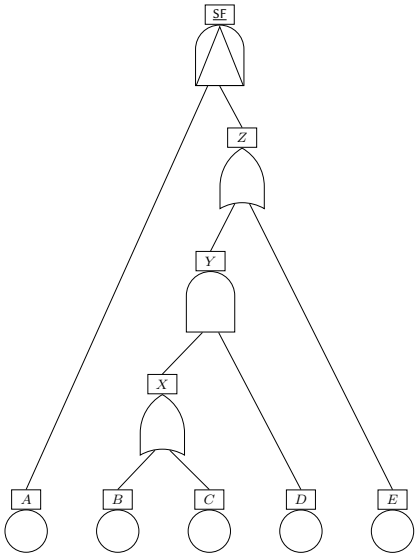
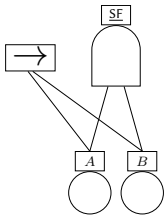
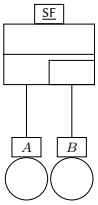


Figure 3.25.: DFT in which an or over-approximates the system performance.



(a) A DFT with sequence enforcing.



(b) An equivalent DFT using cold spares.

Figure 3.26.: Using a cold spare gate for sequence enforcing.

Example 3.32. Consider the DFT in Figure 3.26a. We observe that the system fails after A and B have failed, and that the failure of B can only occur after the failure of A. The DFT in Figure 3.26b is equivalent. The spare gate ensures that B can not fail as long as A has not failed, and only after A and B have failed, the spare gate fails. ▲

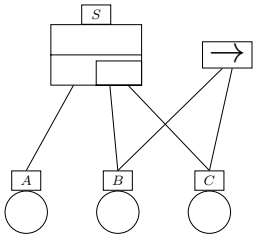
The idea is that the activation mechanism is able to enforce a sequence of events, by assuming cold standby. However, we identify three situations in which such a replacement is not appropriate. We think that the first two are due to limited expressive power of spare gates w.r.t. sequence enforcing, while the last seems to indicate that other ways of modelling yield equivalent DFTs.

The first situation is mainly due to the exclusive claiming of spares. This prevents succesfully combining spare-gates and sequence enforcers.

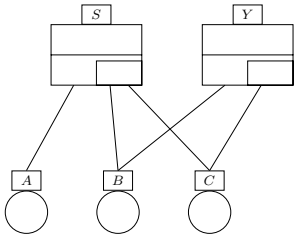
Example 3.33. Consider the DFT depicted in Figure 3.27a. Let B and C be modules with in passive standby. By using the sequence enforcer, we restrict C such that C can only fail after B. Please notice that A does not have to fail first, as B and C are in passive standby. In Figure 3.27b, we depict the DFT after the transformation as described in Example 3.32. We notice that this DFT has, besides questionable syntax, also two semantic issues. First, the claiming of B by the spare gate Y prevents S from ever claiming B, and second, C requires two different dormant failure distributions, as it is both in cold and in passive standby. ▲

The second situation yields different results in case one of the restricted inputs is a gate.

Example 3.34. Consider the DFT depicted in Figure 3.28a. It is similar to the DFT for mutual exclusion, as given in Figure 3.11 on page 37. If we use the scheme from Example 3.32, we obtain the DFT depicted in Figure 3.28b. Now, initially, as A did not occur, the spare module with basic elements B, C is not active. Under the assumption of cold standby, B and C are thus disabled. However, in the original DFT, only the failure propagation through D is restricted. Therefore, both

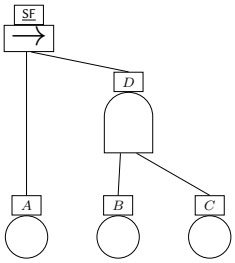


(a) A sequence enforcer and a spare gate.

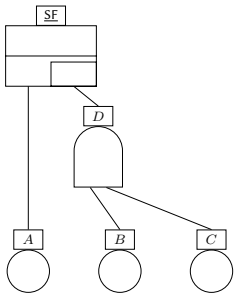


(b) Erroneous attempt with two spare gates.

Figure 3.27.: Combining sequence enforcers and spare gates.



(a) A sequence enforcer with a gate as restricted input.



(b) An erroneous attempt with cold spares.

Figure 3.28.: Sequence enforcers with a gate below restricted inputs.

B and C are initially enabled. Only the sequences BCA and CBA violate the restriction imposed by the sequence enforcer. ▲

To see why it is impossible to simulate the DFT depicted in Figure 3.28a with spare-gates, we notice that with spare gates, we can only activate a module once. Modules which are active are never deactivated. Reconsidering Example 3.34, we notice that for the sequence BAC, C is active initially, and deactivated after the occurrence of B. After A has occurred, C is activated again. This is not possible to emulate with spare gates.

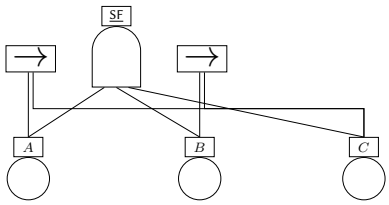
The third case considers multiple sequence restrictions.

Example 3.35. In Figure 3.29, we display issues when multiple sequence enforcers are used. The DFT in Figure 3.29a ensures that C fails only after A and B have failed. A straight-forward approach would be to use cold spares instead of sequence enforcers, depicted in Figure 3.29b, using the same scheme as in Example 3.32. However, this is incorrect due to the claiming behaviour induced by spare gates. Consider the failure of first A and then B. The failure of A causes the spare to claim C, which means that the failure of B causes the corresponding spare gate to fail, as it cannot claim C anymore. However, this scenario can be resolved differently. As indicated by the phrasing above, the DFT in Figure 3.29c is equivalent to the DFT in Figure 3.29a. This DFT can then be translated as expected to Figure 3.29d. ▲

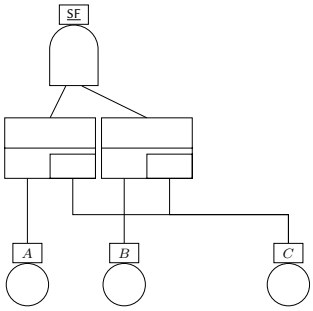
Although in many situations, cold spares are not suitable to directly model sequence enforcing, sequence enforcers can be emulated in several situations.

Notice that the straight-forward application of sequence enforcers restricts the occurrence of a failure to happen only after certain failures. If, however, we want to restrict a failure such that it only occurs before some other failure, then a solution using sequence enforcers requires a less-straightforward solution, which is similar to the construction for mutual exclusion, cf. Figure 3.11 on page 37. Instead of this construction, priority gates may yield a less verbose solution. In the following example, based on a case study from [WP10], we show both approaches.

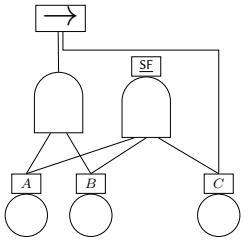
Example 3.36. Consider a primary system which is monitored with a sensor, which can fail (sensor fails, A). If the sensor detects problems (wrong data, B) in the primary system, the system is reset. If (reset error, C) has occurred, the system reset yields a dangerous situation. We assume that the combination of wrong data and the reset error leads to a system failure. We depict a corresponding



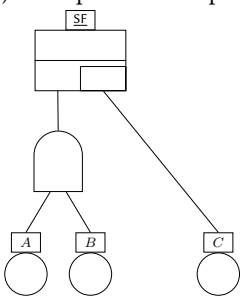
(a) Using sequence enforcers.



(b) Attempt with cold spares.



(c) Using alternative representation with sequence enforcers.



(d) Equivalent solution with cold spares, directly obtained from the alternative representation for sequence enforcers.

Figure 3.29.: Comparing sequence enforcers and cold spares for multiple sequence enforcers.

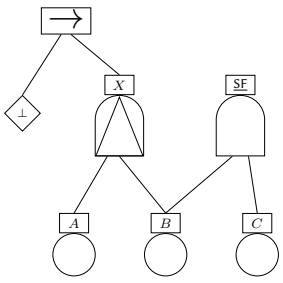
DFT using a sequence enforcer in Figure 3.30a. The occurrence of first A and then B is prevented, as it would cause a failure of X, which is not allowed due to the sequence enforcer.

In [WP10] however, this was modelled analogously to the DFT presented in Figure 3.30b.

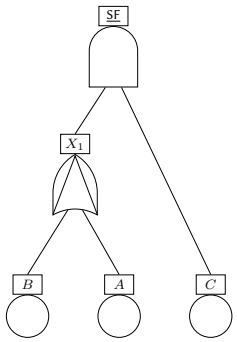
Using priority-or, we model that the fallback system is initialised if (B) occurs, provided the sensor did not fail first, as then, the wrong data issue is never detected. In other words, POR(B, A). The initialisation of the second system is correctly modelled, however a more direct solution would have been to add a sequence enforcer excluding from any detection from a failed sensor. ▲

3.3.4.9. Behaviour of failed spare gates

Some semantics allow functional dependencies to have dependent gates instead of only dependent events. Gates then fail either when the type-dependent condition on their successors is fulfilled, or if triggered. In semantics where dependent gates are not allowed, a simple construction, depicted in Figure 3.31, is used to model the same behaviour. Moreover, semantics which allow dependent gates may use the same internal representation.

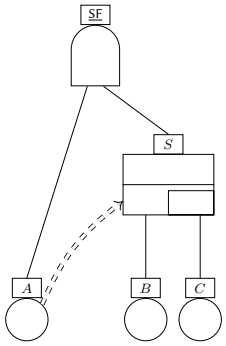


(a) Using a sequence enforcer.

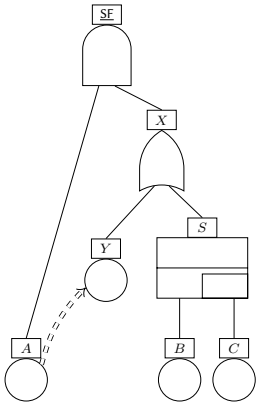


(b) Alternative with a POR.

Figure 3.30.: Emulating sequence enforcers using priority-or.



(a) Functional dependent gates.



(b) Representation via extra event.

Figure 3.31.: Internal representation of dependent gates.

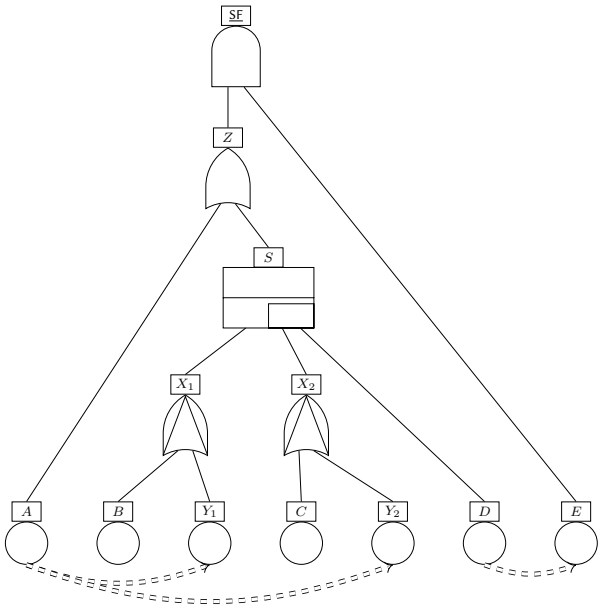


Figure 3.32.: Deactivation of spare modules via POR.

This naturally leads to a situation in which the spare gate is expected to not claim any further spare modules. This construction is a specific case, where even though the spare gate itself has not yet failed, any future failure of the spare gate has no influence on the occurrence of the system failure. Thus, one would expect that the spare gate does not "care about" failing, i.e. no new modules are claimed to prevent the failure. In terms of Example 3.26 on page 44, one would not change batteries of a broken radio. However, many semantics do not support such a deactivation of a spare gate, neither due to failure forwarding nor due to the "don't care"-scenario. We notice that this can be circumvented, again by an overly complex DFT. The method used closely resembles the usage of the POR in Example 3.36 on page 52.

Example 3.37. The DFT depicted in Figure 3.32 shows a DFT with a spare gate S, which does not claim any further spare modules after the failure of A. This is realised by bringing the spare modules in an infallible state. Concretely, the occurrence of A is forwarded to the special events Y_1, Y_2, whose failure prevents any future failure of X_1 and X_2. Therefore, the spare gate S will never claim any new spare module after the occurrence of A. As a consequence, D is never activated after an occurrence of A. ▲

Remark 12. The example shows a solution for only one spare gate. In case multiple spares share spare modules, the DFT gets more complex, as we need POR gates for each spare gate and additional

gates and basic events to keep track of the claiming state. As such a construction depends on the precise semantics[1], we do not present the general case here.

3.3.4.10. Evidence

Often, from a generic fault tree, certain scenarios are derived in which certain states (or faults) are assumed in the system. This is particularly interesting when considering survivability of a system, i.e. the behaviour of the system given a specific bad state. Using so called evidence has a couple of advantages, as described by Haverkort in [Hav14]:

> “Given Occurrence Of Disaster (GOOD) models start with a disaster, hence, there is no need to model the "failure process" or the "disaster probability."
>
> GOOD models avoid:
>
> - estimating rare-event disaster probabilities
> - estimating attack success probabilities
> - stiffness in model evaluations
>
> ”

A more detailed discussion of this is found e.g. in [CH05]. The current version of DFTCalc (cf. Section 3.5.5 on page 69) supports evidence.

Evidence can be naturally modelled by using constant fault element in a DFT. However, it requires semantically well-defined handling of the multiple "simultaneous" failures at initialisation. This is especially important if primary modules of spare gates fail, as it may lead to an underspecified state w.r.t. the spare usage of the different spare gates.

Summary

For designers of DFT tool support, the intricacies presented above show some of the possible choices - and consequences - for the numerous decisions for which the literature has not yet agreed upon a common solution. Furthermore, the intricacies show the expressive power of the DFTs, including the possible workarounds for situations which are not directly supported by DFTs. Such workarounds lead to even harder to understand instances, but also to a class of DFTs where from the designers view, it is though to make assumptions about the structure of the DFT. The list of intricacies also yields welcome support for examining and comparing existing semantics in the next section.

3.4. Case studies using DFTs

In this section, we present existing DFTs from the literature. We aim to show the wide variety of features. We therefore include instances of the following families of DFTs.

- The DFT example from the NASA handbook on fault trees [VS02] (HECS) and the railroad crossing example from the Arrangeer project (RC).
- The most commonly used benchmarks for tools (MCS, FTPP, CAS).
- Two benchmarks with an interesting structure w.r.t. FDEPs (AHRS, NDWP)
- Example for conditional claiming of spares (3S).
- A case study of a plant (SAP), of a airplance controller (MAS), and of a maritime propulsion system (FDS).
- Generated DFTs from an architecture language (SF).

We choose to not to include the following DFTs:

[1] In fact, whether this construction is possible in the general case also depends on the chosen semantics.

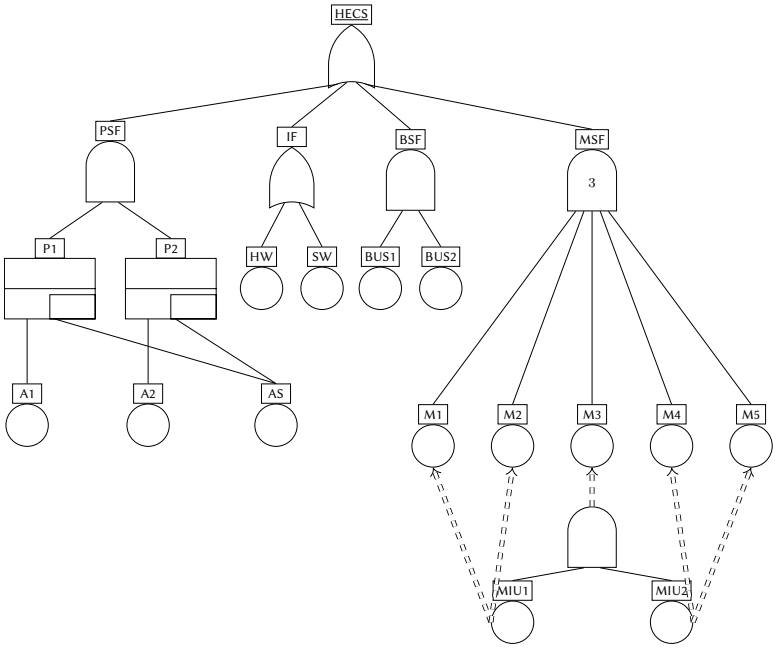


Figure 3.33.: Hypothetical Example Computer System DFT.

- Examples given in this thesis, as they were mostly crafted to show particular features.
- FTHPA [DBB92] as the structure is covered by FTPP and MAS, and the used modelling seems outdated.
- Cascaded Pand [BD05b; YY08] benchmarks, as the structure of these synthetic benchmarks is also found in SAP.
- The controller from the automtive industry given in [Sch09], as it is too large to present here. The structure of the dynamic parts is also captured by other benchmarks.
- Fault trees for railroads (MOVARES) as discussed in [GKSL+14], as they are too large and details are confidential.
- The X2000 Avionics system architecture DFT as presented in [TD04], as it only features one spare-gate in the overall DFT.

It is important to notice that the large case studies we exclude here are largely static, i.e. only a fraction of the fault trees contain dynamic gates. These are interesting for rewriting, but do not suit our discussion of features present in DFTs.

After the introduction, we give a short summary of the features and structures we found in the existing benchmarks.

The failure rates are not in the scope of this section. Full details of models used for benchmarking are found in Section 6.3.1 on page 165.

3.4.1. Hypothetical Example Computer System

The Hypothetical Example Computer System (HECS) is a system described in [VS02] as an example of modelling with DFTs. It features a computer system consisting of a processing unit, a memory unit and an operator interface consisting of hardware and software. The subsystems described above are connected via a 2-redundant bus. The processing unit itself consists of two processors and an additional spare processor which can replace either of the two processors. The processing unit requires one working processor. The memory unit contains 5 memory slots, with the first three slots connected via a memory interface and the last three connected via another memory interface. Memory slots either fail by themselves, or if all connected interfaces have failed. The memory unit requires 3 working memory slots. We depict the fault tree in Figure 3.33.

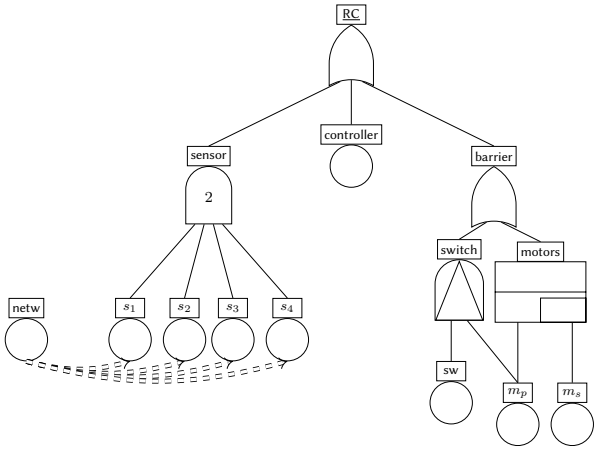


Figure 3.34.: Railroad Crossing DFT.

3.4.2. Railroad crossing

The Railroad Crossing (RC) is a DFT which gives an example DFT for failures at level crossing, introduced by Guck *et al.* in [GKSL+14]. The railroad crossing fails if one of its modules fail. There are three modules.

- The barrier, which consists of a primary motor with a cold spare motor, as well as a switching unit for changing the motor. The barrier fails if either both motors fail, or if the switching unit fails before the failure of the first motor.
- The controller, which is considered a basic component.
- The observation module, consisting of four sensors. These sensors communicate over a network. In case the network fails, the sensors are considered to have failed. The observation module is operational as long as three sensors are operational.

The DFT is depicted in Section 3.4.2.

3.4.3. Multiprocessor Computing System

The Multiprocessor Computing System (MCS) is another model for a computer. The original source considered modelling dependability via Petri nets [MT95]. It considers computing modules (CMs) consisting of a processor, a memory unit and two disks. A computing module fails if either the processor, the memory unit or both disks fail. In the presented MCS, there are two computing modules connected via a bus. Additionally, a memory unit is connected to the bus, which can be used by both computing modules in case the original memory module fails. The MCS fails, if all CMs fail or the bus fails.

Translations into DFTs are given by both Crouzen [Cro06] and Montani *et al.* [MPBC06]. Crouzen changed the original model by adding the assumption that usage of the spare memory component is exclusive, i.e. only one computing module can use the spare memory component. Montani *et al.* additionally added a power supply, whose failure causes all processors to fail with a certain probability. Based on the model, we assume this probability must be one. The DFT is depicted in Figure 3.35.

Arnold *et al.* [ABBG+13] took the version as presented by in [MPBC06] and scaled it to four computing modules, by assuming two independent groups of computing modules with independent spare memory and power supplies. Another variant is given by Codetta in [Cod05]. Here, it is assumed that the two disks are divided into a primary and a backup disk. Moreover, a third computing module is added.

3.4.4. Cardiac Assist System

A fault tree for a system for cardiac assistance is presented by Vemuri *et al.* in [VDS99]. Based upon this, Boudali and Dugan present a DFT for a Hypothetical Cardiac Assist System (HCAS) in

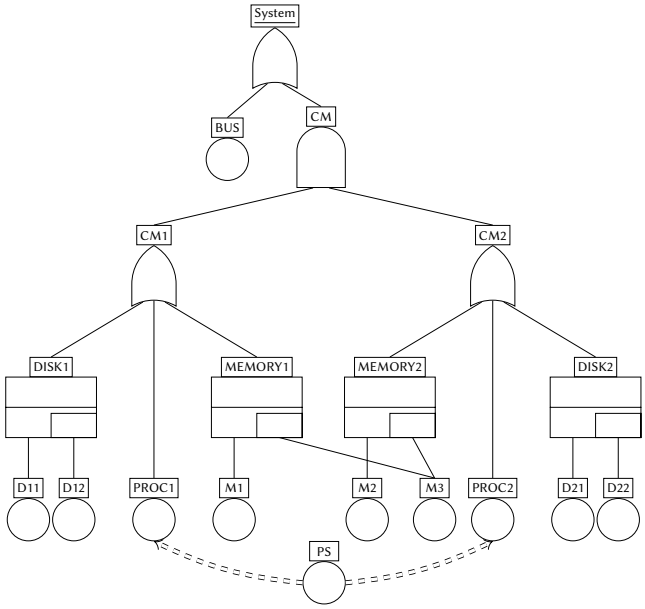


Figure 3.35.: Multiprocessor Computing System DFT.

[BD05b]. The system contains four modules which are all required for correct operation of the HCAS.

- A processing module consisting of a processor with a spare processor in warm redundancy.
- A motor module consisting of two motors of which at least one has to be operational.
- A pump module consisting of two units (L, R). The pump module fails if first L and then R fails. Both L and R consist of a pump. They share a spare pump P_3 which can take over either of them.
- Some switches whose failure propagates to the processors.

Remark 13. We did not find any explanation why the ordering of pump module failures is important. Interestingly, even without any operational pump, the HCAS might still be operational according to the DFT. Besides demonstration purposes - the DFT includes all standard DFT elements, it is possible that the modelled failure for HCAS is not whether the HCAS is operational, but whether it leads to a dangerous situation. In case the pumps fail in reverse order, the system might get into a degraded state which is no direct threat to the patient.

Another DFT inspired by the original description in [VDS99] is given by Boudali *et al.* in [BCS07a]. The Cardiac Assist System (CAS) consists of the same four modules as the HCAS. Two modules have changed however:

- The order in which the pumps fail does not influence the failure of the pump unit anymore.
- Changing a motor requires a switching unit which can fail as well, cf. Example 3.6 on page 32.

We depict the CAS DFT in Figure 3.36.

3.4.5. Fault Tolerant Parallel Processor cluster

The fault tolerant parallel processor (FTPP) cluster is a processor architecture for extra reliability. DFTs for different configurations of such architectures are given by Dugan *et al.* in [DBB92]. They describe a particular instance of an FTPP cluster with 16 processing elements (PE) partitioned into four groups of PEs. Each of these groups contains an additional network element, which is connected to all processing nodes within the group and to all network elements of other groups. On a logical level, the cluster contains 4 triads, each containing three PEs and a spare PE. The cluster has failed as soon as one of the triads fails. Then, with respect to redundancy, three different configurations are considered.

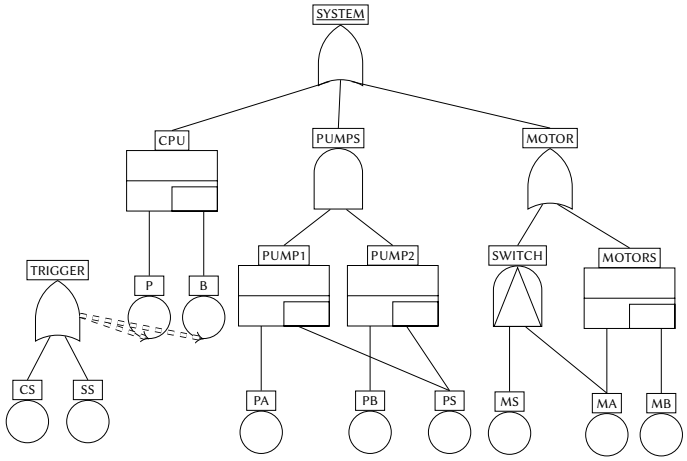


Figure 3.36.: Cardiac Assist System DFT.

Configuration I Configuration I considers the triads to span over the four groups, with each triad consisting of three PEs from the groups one, two and three, and the spare PE from group four. All spare PEs are considered in hot standby. Notice that the representation uses PANDs to model the spare behaviour, as the original source did consider spare modules to always model cold standby.

Configuration II Configuration II considers each triad in a different group, with the spare PEs for each triad in the same group as the rest of the triad. Again, all spare PEs are considered in hot standby. Again, no spare gates were used. A simplified version which does not consider failures of NEs is given by Xiang *et al.* in [XMTY+13].

Configuration III Configuration III is likewise to configuration I, except that the spare PEs are assumed to be in cold standby, and that spare gates are used.

3.4.6. Mission Avionics System

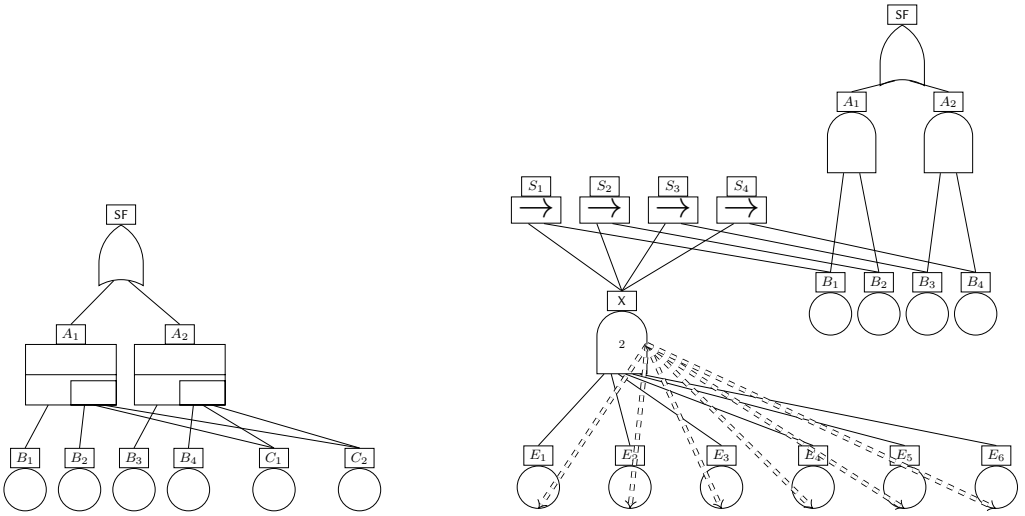
The Mission Avionics System models parts of the electronics architecture for use in military aircraft as proposed by Boeing. Modelling the system as a DFT was done by Dugan *et al.* in [DBB92].

In [DBB92], no hot spares were available. Therefore, modelling hot spares with a shared pool required a work-around. The next example describes the used modelling of pooled spares in hot standby. Notice that it was assumed that each processor has identical failure rates, as otherwise, this type of modelling would not be appropriate.

Example 3.38. Consider a system consisting of two modules, where each module has 2 processors, of which one is redundant. For extra redundancy, two additional processors are given which can replace any of the processors in any of the modules. A processor is replaced as soon as it fails. A direct solution using hot standby is given in Figure 3.37a.

We observe that for the first processor fault to occur, each of the processor faults has an identical effect on the subsequent state of the system. Either, one of the processors in of the modules fails, which leads to claiming one of the processors from the pool, i.e. the size of the pool of available spare processors is decreased by one. Otherwise, one of the processors in the spare pool fails, which also leads to one less available spare processor. For the second fault, the same reasoning holds. After the first two faults, no spare processors are left and each processor which fails causes a spare gate to fail.

In [DBB92], the same behaviour is modelled by a DFT as depicted in Figure 3.37b. Instead of six basic events, ten are given. The six basic events labelled $E_1 \dots E_6$ describe the processor faults of any of the six processors under the assumption that at most one processor has failed. The basic events $B_1 \dots B_4$ describe processor faults of the processors which are assigned to one of the modules and cannot be replaced by spare processors anymore. The sequence enforcers ensures that



(a) Using spare-gates to model spare sharing.

(b) Work-around with sequence enforcers.

Figure 3.37.: Pooled spares remodelled.

faults $B_1 \ldots B_4$ cannot occur before the spare pool is empty. The functional dependencies reflect that none of the processors can fail anymore as being spare elements after the pool is empty. ▲

In Figure 3.38, we depict the version with spare gates, as given by Vemuri *et al.* in [VDS99]. Besides general hardware (MEMory, DataBus, etc.), the model features a large number of general purpose processing units with dedicated task, with some spare processing units capable to replace any of the failed units.

3.4.7. Active Heat Rejection System

The Active Heat Rejection System (AHRS) DFT is introduced by Boudali and Dugan in [BD05a]. Heat rejection systems are commonly found in air- and spacecrafts. The DFT, depicted in Figure 3.39 models a system with two redundant thermal units containing a pump unit. Each pump unit has a dedicated spare pump in cold standby. Moreover, the two pump units share an extra spare pump. The primary pumps are connected to a power generator, which moreover powers the dedicated spare pump of the other power unit, respectively. The shared spare pump has a dedicated power generator.

3.4.8. Non-deterministic water pump

The Non-Deterministic Water Pump is a synthetic case study by Boudali *et al.* [BCS07a] to show the support for non-determinism in the presented approach. We depicted the DFT in Figure 3.40 on page 62. The system consists of two pump units (P_1, P_2) each consisting of a regular pump (A_1, A_2) and a backup pump system (S_1, S_2). The backup pumps systems share a spare pump (B_3). The backup pumps primary modules (B_1, B_2) have a common cause failure (X). The assumed non-determinism is due to the spare race in case X fails.

3.4.9. Sensor-filter

The Sensor-Filter benchmark (SF) is obtained from the Compass tool set [BCKN+09]. Given an AADL[1] model of a system, the tool set searches for combinations of basic faults which lead to the predefined configurations in the given system. These combinations are represented in form of a DFT, cf. Bozzano *et al.* [BCCK+10]. The sensor-filter benchmark is a synthetic example, which contains a number of sensors and filters which are connected to each other. In Figure 3.41 on page 62, a DFT for a particular instance with one sensor and one filter and four fault configurations is given.

[1] Architecture Analysis & Design Language, see [FGH06]

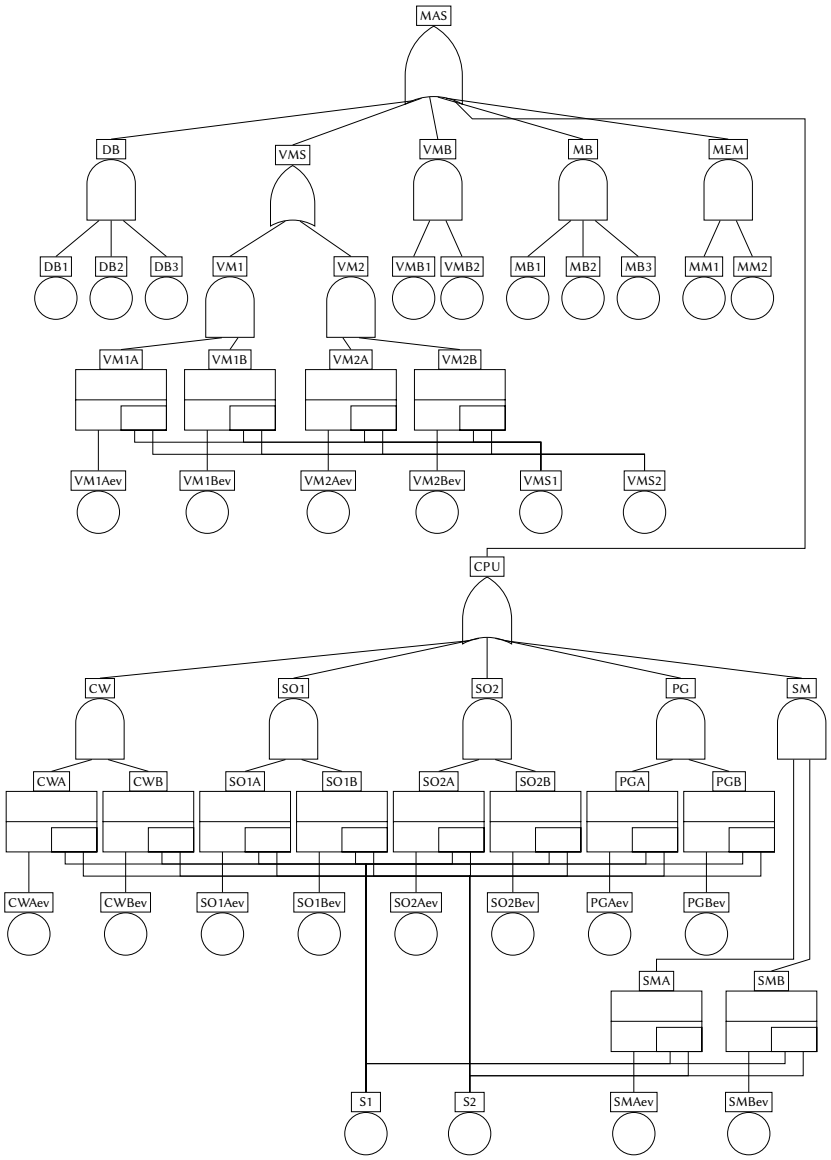


Figure 3.38.: The Mission Avionics System DFT.

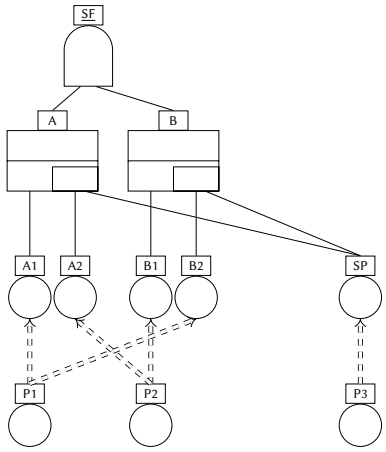


Figure 3.39.: Active Heat Rejection System DFT.

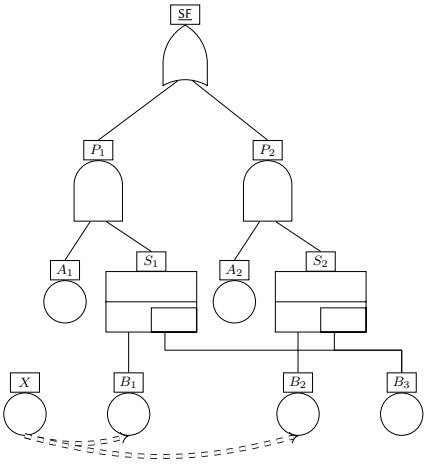


Figure 3.40.: Non-Deterministic Water Pump DFT.

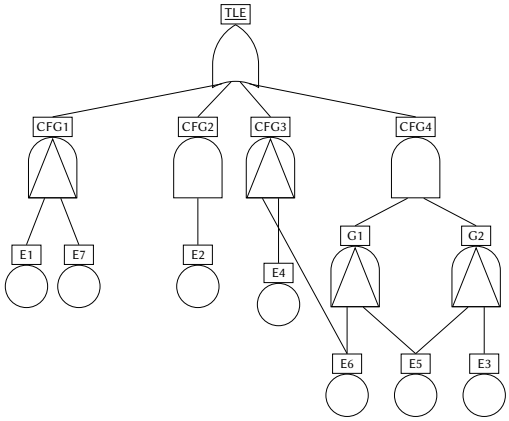


Figure 3.41.: Sensor-filter DFT

Remark 14. Based on the description of the system, we doubt that the usage of PAND is intended. The PANDs originate from the translation of failures which are reached via a sequence of faults. The latter faults are, according to the description given, only enabled after the former. This event restriction is not modelled in the given DFT. According to the given DFT, the latter events are enabled from the start and their failure brings the system in a fail-safe state, cf. Example 3.16 on page 37.

3.4.10. Section of an alkylate plant

A Section of an Alkylate Plant (SAP) was modelled as DFT by Chiacchio *et al.* in [CCDM+11b], based on an undescribed[1] fault tree from an unspecified safety report. The PANDs are inserted "as the alarm equipment considers the time of occurrence of a fault." The spare gates is used to model cold standby of a spare battery. Notice that two of the basic events (BE1 and BE3) are originally assumed to be discrete probabilities. This could be modelled using either PDEP gates or by analysing the four DFTs obtained by replacing both BE1 and BE3 with both constant failures and fail safe elements and combining the obtained measures accordingly.

Remark 15. We observe that assuming BE1[2] and BE3 to have a discrete probability leads to the following unmentioned behaviour. In those cases where BE1 holds, it holds from time $t = 0$, and so does IE6 and IE4. Then, IE1 -and thus, the system- can only fail if IE3 holds from $t = 0$. This can only happen if also BE3 holds, as the probability that IE9 holds from $t = 0$.

Here, we give the fault tree and assume constant failure rates for BE1 and BE3.

[1] Only a technical specification of the basic events is given.
[2] According to the original source this is a "human error."

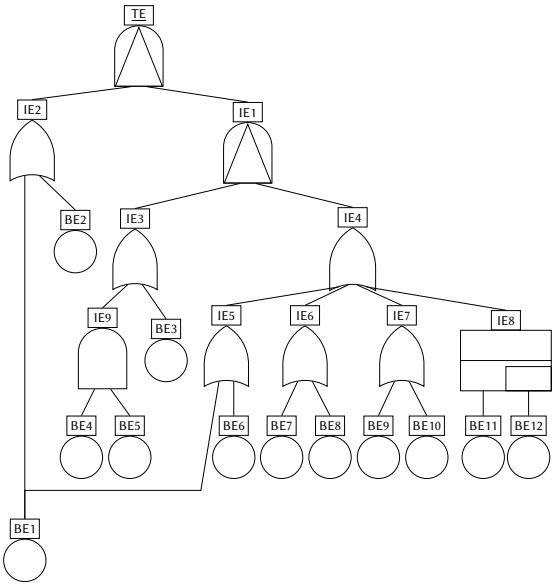


Figure 3.42.: Section of Alkylate Plant DFT.

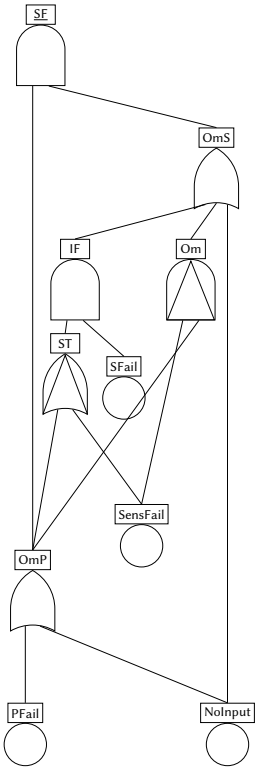


Figure 3.43.: Simple Standby System DFT.

3.4.11. Simple Standby System

The Simple Standby System (3S) is a benchmark, for fault trees expressed in Pandora, described by Walker and Papadopoulos [WP10]. It features a monitored system which in case a sensor indicates errors in the primary system switches to the standby system. We give the description of the system, following the phrasing of [WP10] but changed the ordering to reflect the top-down approach.

- System failure is caused by a combination of both Omission- Primary (OmP) and Omission-Standby (OmS).
- Omission of Standby (OmS) occurs if it has been triggered and it suffers an internal failure (IF), if it does not get triggered (Om), or if there is no input (NoInput).
- An internal failure occurs if the standby has been triggered and has failed (SFail).
- Standby is triggered (ST) if the sensor detects omission from primary before the failure of the sensor occurs (SensFail).
- Omission of trigger occurs if the sensor fails before or at the same time as an omission of primary.
- Omission from Primary is caused by an omission of input (NoInput) or internal failure (PFail).

The Pandora expression for the fault tree uses gates for simultaneous failures and exclusive PANDs, which is not supported by any of the available tools. However, these expressions are always connected via an or-gate, which yields the inclusive PAND. In Figure 3.43, we give the version based on the inclusive pand-gate, which can be handled by DFTCalc. We notice that the POR gate was orinally supposed to be exclusive, however, a simultaneous failure almost surely does not occur.

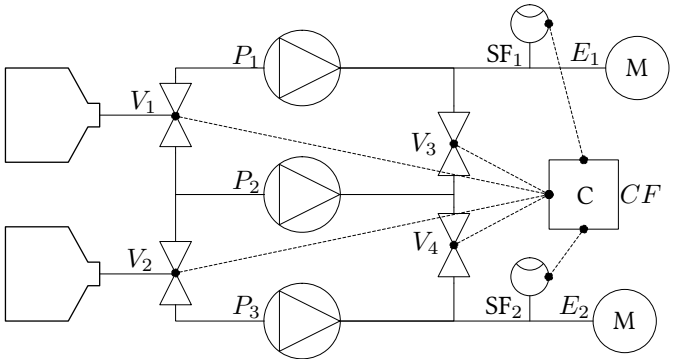


Figure 3.44.: Schematic representation of the Fuel Distribution System.

3.4.12. Fuel Distribution System

A fuel distribution system for a maritime propulsion system is given by Edifor *et al.* in [EWG12].

The system, depicted in Figure 3.44, consists of two engines. Each engine has its own fuel tank. Between the tank and the corresponding engine, a pump is used to pump the fuel from the tank to the engine. A flowmeter in front of the engine monitors the fuel flow. In case it is interupted, a controller is informed. The system has a spare pump, which is connected to both tanks and both engines. In case the controller is informed of a missing fuel flow via either of the sensors, the system is reconfigured such that the spare pump replaces the broken pump. This is done via two valves, respectively. One of the valves is in front of the primary pump and reroutes the fuel to the spare pump, the other behind the spare pump to reconnect the engines fuel supply with the spare pump. Please notice that the spare pump can replace either of the pumps, but not both.

The system fails if either of the engines fails. An engine fails if it either has an internal failure, or has no fuel supplied. No fuel is supplied if the primary pump fails (internally) and the primary pump is not replaced by the external pump. The primary pump is not replaced if

- the spare pump successfully replaced the other pump, or
- the corresponding sensor has failed before the failure of the pump, or
- the controller has failed, or
- one of the two corresponding valves have failed before the failure of the pump.

The spare pump is assumed to be in cold standby, so it can only fail after it is put into operation. The failure of valves means that they are stuck, i.e. after their failure, they cannot be used during reconfiguration. Otherwise, their failure has no impact on the system.

Remark 16. The authors present also a fault tree, given as a Pandora expression. We notice that the given fault tree does not match the description of the system, e.g. a failure of first pump 1 and then valve 3 leads to a failure of engine 1. Moreover, the spare pump is described to be in cold standby, however, this cannot be modelled in Pandora, cf. also Section 3.3.4.8 on page 50.

In Figure 3.45 on page 66, we present a DFT based on the original fault tree but adapted such that it matches the given description. We do, however, assume P3 is in hot standby, as it is in the original fault tree.

We describe the fault tree here only for engine 1. Either the engine fails internally (IE1) or there is no fuel delivered to the engine (NFE1). No fuel is delivered either with pump 1 and pump 3 (NFE1WP3) or without pump 3 (NFE1NP3).

Let us first unfold the case with pump 1 and 3: No fuel is delivered if pump 1 and pump 3 have failed and engine 1 has claimed pump 3 (E1P3). Engine 1 claims pump 3 in two different scenarios. Either pump 1 has failed before pump 2 (P1BP2) and nothing which prevents reconfiguration of pump 1 to pump 3 (NP1CP3) has happened before that, or pump 2 has failed without claiming pump 3 before pump 1 (P2TP1NC) and then pump 1 failed and nothing which prevents reconfiguration of pump 3 to pump 1 (NP1CP3) has happened before that. Pump 2 fails to claim pump 3 either if pump 1 claimed it first (which is not applicable in the case of P2TP1NC, or if something which prevents reconfiguration of pump 3 to pump 2 failed before pump 2.

Failures which prevent engine 1 from claiming pump 3 (NP1CP3) are a failed pump 3 or a reconfiguration error (RE1), which is either a controller failure, a sensor failure of the flowmeter, or valves 1 or 3 failing. For engine 2, this is analogous.

We now consider the case with pump 1 and not pump 3: No fuel is delivered if pump 1 has failed an the engine cannot claim pump 3 anymore (NE1P3). This is the case if pump 3 has failed, or engine 2 has claimed pump 3 (E2P3, analogous to the case E1P3) or a reconfiguration unit does not work anymore (RE1).

3.4.13. A brief discussion of the benchmark collection

Above, we've presented a collection of benchmarks. Here, we briefly review the benchmark collection.

We observe that in all benchmarks, the spare modules are basic events, none of the benchmarks contain subtrees, or even nested spares, as spare modules. This seems partly due to the fact that subtrees for spare modules are not supported by the original tools which used the benchmark.

- In, e.g. HECS (Figure 3.33 on page 56), memory failures are much deeper developed than processor failures, which might be due to - now outdated - restrictions on spare modules.
- In, e.g. CAS (Figure 3.36 on page 59), the primary modules of one of the spares is not independent. While this is not problematic in this particular case, nesting such a DFT below another spare leads to undefined behaviour[1].
- In, e.g. AHRS (Figure 3.39 on page 61), the shared spare pump and its power supply are connected via an FDEP, but the power supply could be also be a spare module together with the spare pump. This would also allow the power supply would be only activated upon usage, likewise as in Example 3.29 on page 46, without resorting to extended semantics for FDEPs.

Spare races can occur in MCS (Figure 3.35 on page 58) and FTTP. Resolving the non-determinism due to claiming is potentially problematic in these benchmarks - depending on the assumed failure rates and the usage of evidence. In many other benchmarks, the common cause failures directly lead to system failures, which makes the resolution of spare races obsolete.

Please, notice that although [VS02] explicitly described the usage of FDEPs to resolve cyclic dependencies introduced by feedback loops, none of the benchmarks actually have such loops. Therefore, the lack of causal ordering as discussed in Section 3.3.4.2 on page 41 has no effect on the obtained results. Ordered failure combination, has only an effect on the result of the SAP and FDS benchmark (Figure 3.42 on page 63).

To the best of our knowledge, none of the benchmarks encodes a non-coherent system, so no or-gates are used to model xor-gates. Changing the behaviour of failed spare gates would change some results, e.g. in the MCS benchmark (Figure 3.35 on page 58). Likewise, the comments about the unclear behaviour w.r.t. evidence also apply on that DFT.

Furthermore, sequence enforcers were only used in older versions of some DFTs to model the spare management. With tool support for the warm spare gate, sequence enforcers are not present anymore. On the other hand, we've seen that in — at least — the sensor-filter benchmark, sequence enforcers would allow a more accurate modelling of the system behaviour. The usage of cold spares to restrict the sequencing is not flawed in any of the benchmarks.

We observe that (correct usage) of priority gates is mostly used to model reconfiguration, as it gives more freedom then spare gates. On the other hand, without spare gates, warm spares can only be modelled via extra basic events and sequence enforcers.

We notice that all presented benchmarks, except MAS and FTPP, were presented to accompany the feasibility of some particular approach. The DFTs are therefore often compact and have only a small static fragment. Based on fault trees in the Arrangeer and error models in the Compass project, we claim that DFTs for many systems are indeed largely static, i.e. the vast majority of elements is static, presumably even the vast majority of subtrees. This is substantiated by the fault trees presented in [Sch09; GKSL+14; TD04]. Most descriptions of DFTs which accompany tools for DFT analysis do not match the guide lines for hierarchically constructed DFTs — it seems that many constructs are crafted to match the system, but not following a hierarchical approach. Some

[1] The formalisations in Section 3.5 on page 67 all exclude sharing successors of spare gates.

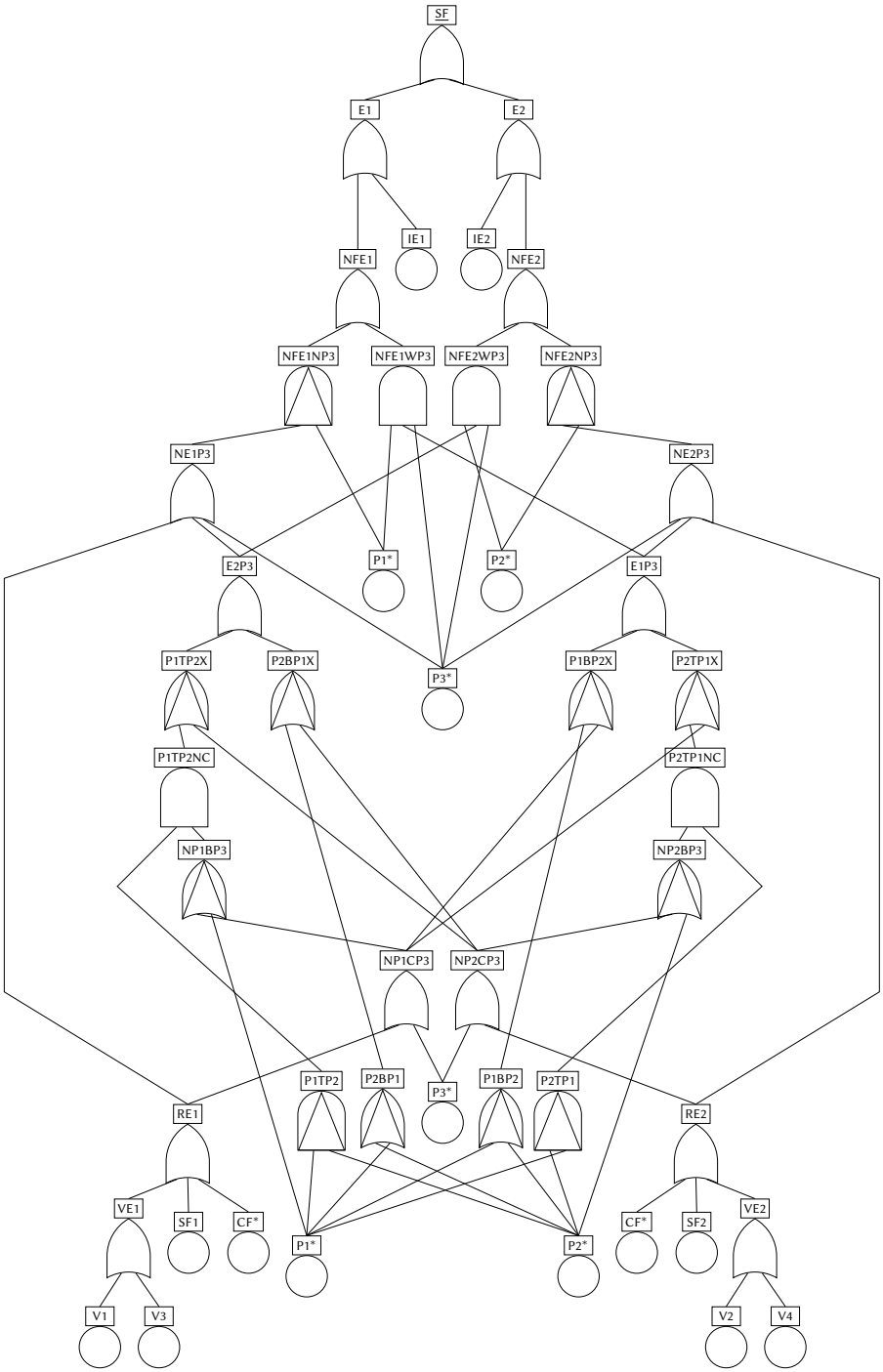


Figure 3.45.: Fuel Distribution System DFT.

of these benchmarks are thus of limited value when it comes to assessing the practical relevance of simplification in Chapter 5 on page 113.

3.5. Formalising DFTs

In this section, we consider and compare several existing formalisations of DFTs.

It is important to notice that upon the introduction of DFTs (e.g. DFTs are used e.g. in , they where not formalised. This has lead to an unclear meaning of specific fault trees, as outlined in [CSD00]. Since this initial formalisation, several others have been introduced which are not fully compatible to each other. We discuss eight different formalisations in greater depth. A tabular comparison of specific features is given afterwards, in Table 3.2 on page 70. We do not include the formalisations as used in the two Monte-Carlo based approaches presented in [BNS09] and [CCDM+11a], and the definition as given in [RS14]. Attempts for using minimal cut sequences as presented in [TD04] and in [LXZL+07] are excluded as they these are not suitable to describe the behaviour of DFTs, as discussed in Section 3.3.4.1 on page 40.

3.5.1. Fault tree automaton construction

The first formalisation of the semantics of a DFT is given by Coppit *et al.* in [CSD00]. It gives an operational semantics-style axiomatisation of DFTs, formalised in Z[1]. The semantics provides a notion of a state of the fault tree, which contains information about the order in which elements have failed as well as usage information for the spare gates. Then, for any two states s, s' and a basic event e, for a DFT in state s and the occurrence of e, it is formalised whether s' is a valid resulting state w.r.t. the semantics. Based on this, a *fault tree automaton* is constructed which describes the non-deterministic transition system. Then, for a given fault tree automaton, the underlying CTMC is defined, which enables us to calculate the reliability function of a fault tree by computing it on the underlying (deterministic) CTMC.

The formalisation of PANDs is inclusive. FDEPs cause immediate propagation of the failure to the dependent events. Triggers are allowed to be subtrees, while the dependent events should be basic events. Spare gates require that their successors are basic elements. All such basic events are required to have only functional dependencies, spare gates or sequence enforcers as predecessors. Sequence enforcers are included in the most general form. In case of a spare race, non-determinism occurs. Please notice that no notion of causality is included. In the translation to the CTMC, the non-determinism is resolved by a uniform distribution as described in Section 3.3.4.3 on page 43.

Tool support in Galileo was presented in [SDC99]. Some of the underlying algorithms were presented in [MCSD99].

3.5.2. Reduction to Bayesian Networks

A popular method to support quantitative analysis of (dynamic) FTs is based on a reduction some kind of Bayesian Networks (BN) [Pea88]. We consider the reduction to Discrete Time Bayesian Networks (DTBN) in [BD05b] and the reduction to Continuous Time Bayesian Networks (CTBN) in [BD06] both by Boudali and Dugan, as well as the reduction to Dynamic Bayesian Networks (DBN, [Gha98]) by Montani *et al.* in [MPBC06; MPBV+06; MPBC08]. The underlying idea is to introduce random variables for each element in the fault tree. Random variables representing gates are conditionally dependent on the random variables representing the children. Notice that no cycles introduced by FDEPs are allowed in the DFT, as this would yield a cyclic BN. Basic events are multi-valued variables, which encode not only whether they've failed, but also whether they are active. This enables the integration of warm-standby.

Tools for Bayesian network analysis are widely available. The reduction to Bayesian Networks allows several additional analyses on Fault trees, e.g. the *most likely explanation* analysis which is not treated in other papers.

[1] A formal language based upon Zermelo-Fraenkel set theory[Spi92].

3.5.2.1. Reduction to Discrete or Continuous Time BN

The reduction to a DTBN as described in [BD05b] divides the interval from $t = 0$ to $t = T$, where T denotes the mission time, into n intervals. Failure events are then during an interval $0 \leq i \leq n$, instead of at a time point $0 \leq t \leq T$. Notice that for $\lim_{n\to\infty}$, the DTBN is equivalent to the CTBN described in [BD06]. Obviously, the discretisation introduces some inaccuracies. On the global level presented here, besides this inaccuracy, the two formalisms are equivalent.

Each gate is represented by a random variable. Conditional dependencies with all children are given. For static gates, these conditional rules are directly derived from the truth table for these gates. PANDs are assumed to be inclusive. FDEPs directly cause the failure of their dependent events. The triggers may be subtrees, while the dependent events are assumed to be basic events. Spare gates are assumed to have only basic events as successor. Sharing spares is not explained in the paper. Moreover, common cause failures for spare modules is not handled.

3.5.2.2. Reduction to Dynamic BN

The encoding to a Dynamic Bayesian Network discretises the time, but instead of slicing, it assumes discrete time points for each event.

For each element in the DFT, a DBN is introduced, which are merged into a single DBN afterwards. During the merging process, the conditional probability tables are merged. For this process, it is assumed that the conditional failure probability is equal to the maximum conditional failure probability in the two merged nodes, given any condition - which introduces an error but yields smaller tables. For the PAND, an additional variable is introduced which keeps track of the ordering. PANDs are considered inclusive. FDEPs are extended to PDEPs. Failures are instantaneously propagated. Warm spare gates assume basic events as children. The behaviour in case of a spare race is unspecified. Sequence enforcers are not included. Tool support was included in the tools DBNet[MPBC06] and Radyban[MPBC08]. However, none of these is publicly available at the time of writing.

3.5.3. Reduction to Stochastic Well-formed Petri Nets

Stochastic Well-formed Coloured Nets (SWN) [CDFH93] are an extension to Petri nets. Bobbio *et al.* [BFGP03] reduce parametric fault trees to SWNs. Parametric fault trees are SFTs with subtree-replication, which yields a smaller state space by exploiting symmetry. This approach was expanded to DFTs by Bobbio and Codetta in [BC04]. Moreover, repair via so called repair-boxes was introduced. We only discuss ordinary DFTs here.

For each element in a DFT, a small Petri net is given, which has input-places and an output-transition. That is, each DFT element operates based on the presence of tokens in its input places. A failure of the gate causes a transition to fire which then places tokens in any predecessors of the DFT. To compose a SWN for a DFT with multiple elements, the inputs and outputs are merged according to the structure of the DFT.

Remark 17. The semantics of a SWN - as described in the references cited in [BC04] - allow only one transition to fire at a time. Therefore, we assume that only one transition fires at once. SWNs allow for priority assignments to select which transition to fire in cases where multiple transitions are enabled. The presented semantics for DFTs however do not mention this.

Gate failures are ordered, as synchronisation between the elements is done via placing tokens, the effect of any element failing is not simultaneously processed by the predecessors. The PANDs are inclusive, FDEPs do not distinguish different dependent events. Both triggers and dependent events are assumed to be basic events. The spare gates assume basic events as successors. Any sharing is implicitly assumed to be amongst symmetric spare gates using identical basic events as spare elements. Thus, non-determinism during claiming is hidden. Warm and cold standby are discussed, but their implementation remains unclear. Sequence enforcers are presented in a general fashion, but their interpretation when putting restrictions on gates is different. Instead of invalidating a sequence, the gates are just delayed until the earlier gates (w.r.t. the sequence enforcer) have failed. Tool support is presented, but not available any more.

3.5.4. Reduction to GSPN

A reduction of DFTs to Generalised Stochastic Petri Nets (GSPN) [MBCD+94] is given by Codetta in [Cod05]. The overall idea is to use a graph transformation for an element-wise reduction to a GSPN. This GSPN can then be reduced to a CTMC using existing algorithms [Bal07].

In a first step, a place for each vertex is added. A marking on such a place means that the element has failed. Each gate is then replaced by a subnet which places a mark in its output place depending on markings in the input places. The standard semantics for GSPN are assumed, gates thus fail ordered.

The static gates are trivially defined. For PANDs, an extra place checks whether the ordering is respected. The ordering is assumed to be inclusive. FDEPs mark all dependent events failed, in a non-deterministic ordering[1]. Warm spares are supported, but neither spare pool sharing nor non-singleton spare pools are handled. The sequence enforcer requires all successors to be basic events. Tool support is described, but not publicly available.

3.5.5. Reduction to a set of IOIMCs

Input/Output-Interactive Markov Chains (IOIMCs) are an extension of Interactive Markov Chains suitable for a compositional design of Markovian processes. A reduction to IOIMCs is described by Boudali *et al.* in [BCS07c; BCS07a; BCS10]. The overall idea is again to define small IMCs for each element in a fault tree, where inputs are encoded as transitions labelled with an input action an the failure propagation is encoded by a transition with an output action. The encoding of the complete DFT is then given by the parallel composition of these IOIMCs. Moreover, spare gates distribute claiming and activation information via extra transitions. It is important to notice that only one transition at a time can fire in IMCs, which means that all elements fail in some order. The order is non-deterministic in the model.

The encoding of the static gates is straightforward. PANDs are non-inclusive (simultaneous failures do not occur), FDEPs propagate their failures of the triggers to the dependent elements[2]. Both triggers as well as dependent elements may be subtrees. Dependent gates are resolved by extra internal transitions which are alike extra basic events connected to an or-gate, as discussed in Section 3.3.4.9 on page 53. Spare gates have independent subtrees as spare modules. Spare gates in spare modules are allowed and follow the late claiming with early failure mechanism. Sequence enforcers are not included in the semantics.

It has tool-support, delivered by Coral[BCS07b], which is now replaced by DFTCalc[ABBG+13]. DFTCalc includes support for evidence, by replacing the gates with a constant failure.

3.5.6. Algebraic encoding

This section is dedicated to a formalisation of DFTs by an algebraic description, as described by Merle *et al.* in [MRLB10; MRLV10; MRL14]. Similar efforts are described by Walker in [Wal09; WP10] and Schilling [Sch09]. However, those formalisations do not include spare gates and is therefore excluded from the discussion here.

Static fault trees are trivially embedded into Boolean algebra. In [MRLB10], the authors use earlier work which extends the Boolean algebra with temporal operators for *before*, *inclusive before* and *simultaneous* to formalise "priority DFTs with repeated events." These are static fault trees with PANDs and FDEPs. "Repeated events" are used to emphasise that the underlying graph is not necessarily a tree. Although the formalisation method supports both inclusive and exclusive PANDs, the authors choose the inclusive variant as it "seems more coherent with the designers' expectations." All failure propagation is immediate. The authors state that including "the concept of 'non-determinism' is hardly acceptable in the engineering practice." Furthermore, the authors give a canonical representation for DFTs in the algebra, which extends minimal cut sets with ordering information, as well as a scheme for deducing the top-level failure distribution given fault distributions. In [MRLV10], the authors use the same algebra for spare gates. The considered spare gates only allow basic events as successors. It is explicitly assumed that the basic do not occur simultaneous, which excludes common cause failures in spare gates. Activation is realised by considering two events - one with a warm and one with a hot failure rate and explicitly excluding

[1] It remains unclear how this non-determinism is resolved for the reduction to a CTMC.

[2] We follow [BCS07c; BCS07a] here

the occurrence of both failures. Sequence enforcers are not included. Constant-failures are not presented, although present in the algebra. The algebraic encoding has not yet publicly available tool-support.

Table 3.2.: Comparing different semantics for DFTs.

	FTA	CTBN	DBN	SWN	GSPN	IMC	AE
Ord. failures	yes	no	no	yes	yes	yes	no
Spare modules	events	events	events	events	events	subtrees	events
Spare races	yes	no	?	yes	yes	yes	no
Pand-gate	$\leq$	$\leq$	$\leq$	$<$	$<$	$<$	$\leq$
Dep. events	BE	BE	BE	BE	BE	BE & gates	BE
Seq. enforcer	yes	no	cold spares	yes	no	cold spares	no
Por-gate	no	no	no	no	no	no[a]	no[b]
PDEP	no	no	yes	no	no	no	no
Replication	event	no	no	subtrees	no	no	no
Evidence	no	no	no	no	no	no	no[b]
Tool	Galileo[c]	no	DBNet;[c] Radyban[c]	DrawNet	unnamed[c]	Coral;[c] DFTCalc	no

Ord. failures: whether the gates fail ordered (either total or partial).
Spare modules: the type of spare modules supported.
Spare races: whether constructions which possibly lead to spare races are supported.
Pand-gate: whether the pand-gate is inclusive or exclusive.
Dep. events: the type of the non-first successors of an FDEP.
Whether sequence enforcers are supported. Cold spares means that SEQ is modelled via a cold spare.
Por-gate: whether por-gates are supported.
PDEP: whether PDEPs are supported.
Replication: What subtrees are allowed to be replicated.
Tools based on the given semantics.
[a] Recently introduced in DFTCalc.
[b] Can be modelled directly within the framework.
[c] Not available at the time of writing.

4. Semantics for Dynamic Fault Trees

In this chapter, we introduce a denotational style semantics for DFTs. The rationale for introducing yet another semantics, as well as the assumptions which led to the version presented in this thesis, are discussed in the first section. The second section contains formal treatment of the semantics of a DFT in terms of a Markov automaton. Based upon these semantics, the earlier mentioned quantitative measures on DFTs are defined using the underlying Markov automaton and some notions of equivalence are formalised. Furthermore, the chapter contains the outlines of state-space reduction technique based on partial order reduction, which is useful to widen the defined equivalence classes. We finish the chapter by briefly discussing some extensions to the presented semantics.

4.1. Rationale

To formalise the simplification of DFTs - the major topic of this thesis - a precise meaning of simplification is required. In this thesis, we choose to require the outcome of the simplification process to be a DFT which is equivalent w.r.t. the aforementioned quantitative measures. Any algorithm which provides this simplification can only be proven correct in the presence of a precise definition of DFTs. The class of DFTs whose semantics are well-defined should contain the common constructs of shared spares and common cause failures. Notice that syntactic restrictions might lead to more complex simplification steps, as the simplification should always yield a well-defined DFT.

Based on the brief review of existing semantics in Section 3.5 on page 67, two candidates among the existing semantics meet the requirement of being concise on a general class of DFTs. First, the operational style semantics discussed in Section 3.5.1 on page 67 and second, the reduction to IOIMC discussed in Section 3.5.5 on page 69. Please recall that these semantics are not fully compatible to each other.

The lack of available tool-support for the semantics for the operational style semantics and the restrictive spare modules, besides the lack of causal ordering, are major drawbacks for the usage of the operational style semantics. For the IOIMC semantics, the ordered failure combination together with the direct translation into IOIMCs yields a complex context in which rules have to be applied. Furthermore, the late claiming yields additional scenarios which have to be handled correctly.

We therefore choose to define semantics, with the following requirements.

- The semantics should yield a qualitative model to ease arguments, as also done in [CSD00]. The qualitative model should reflect the trace-based view also used in Chapter 3 on page 27 in which we discussed the effect of a trace of basic events on a fault tree. The qualitative model should be subsequently translated into a quantitative model upon which the measures can be defined. The quantitative model should support non-determinism.

- As DFTCalc is the only available state-of-the-art tool, the defined semantics should be compatible with the IOIMC semantics on a large and well-defined class, which should comprise most of the case studies in Section 3.4 on page 55.

- Whenever the semantics are incompatible with the semantics of DFTCalc, it should yield simpler semantics. It is beneficial if the semantics are compatible with the operational semantics.

For the definition of the qualitative model, we use a more denotational style of semantics, compared to [CSD00]. Thus, given a state of the DFT and a basic event failing, the next state is defined. For the quantitative model, we choose Markov Automata as they come with native support for non-determinism.

As the IOIMC semantics use ordered failure combination which was embedded in the temporal ordering, we want to include ordered failures. Based on discussion introduced briefly in Section 3.3.4 on page 40, we assume the failure forwarding to be ordered and the failure combination to be instantaneous. This yields semantics which are equivalent to the operational style semantics if no functional dependencies are present. We can enforce compatibility to DFTCalc by using functional dependencies instead of plain failure combination. We assume inclusive pand-gates as used in other existing semantics, and only include inclusive por-gates as syntactic sugar. The more general exclusive por-gates, as well as sequence enforcers and probabilistic dependencies are briefly discussed later in Section 4.5 on page 110. We assume early claiming, as this prevents failure from activation or failure from claiming. The semantics are defined on a large class of DFTs. Some features which are present in a single other semantics have been left out or only touched in Section 4.5 on page 110. These features are: 1. sequence enforcers present in the operational semantics, 2. the probabilistic dependencies present in the Bayesian Network semantics, 3. The sharing of primary spare modules with the top modules. Apart from these, the class of well-defined DFTs in the given semantics is a superset of each set of well-defined DFTs w.r.t. any existing semantics.

4.2. New Semantics for Dynamic Fault Trees

In this section, we formalise the various mechanisms present in DFTs. After the formalisation of DFTs from a syntactical point of view, we formalise failure combination and claiming on a restricted class of DFTs. We define the *state of a DFT* based on a sequence of basic events which have failed. This yields a deterministic description of the next state.

We then cover functional dependencies and the mechanism of failure forwarding (internal events). Resolving functional dependencies may introduce non-determinism, that is, for a chain of failed basic events, there is no unique state of the DFT. We construct a transducer which encodes the state space based on external and internal events and yields a qualitative model of the DFT.

The effects of activation and the quantitative information is added and during the transforming from the functional transducer into a Markov Automaton (MA).

Moreover, we introduce the notion of a policy and describe some syntactic sugar which can be added to ease the modeller's life.

We do not cover sequence enforcers, probabilistic dependencies or por-gates in this section.

4.2.1. DFT syntax

DFT elements have different types. We distinguish between types for inner nodes, called gates, and types for sinks, called leafs.

Definition 4.1 (DFT gate-types)**.** The set Gates of *DFT gate-types* is defined as

$$\mathsf{Gates} = \{\text{AND}, \text{OR}, \text{PAND}, \text{FDEP}, \text{SPARE}\} \cup \{\text{VOT}(k) \mid k \in \mathbb{N}\}.$$

Elements of such types are called and-gate, or-gate, priority-and gate(pand-gate), functional dependency, spare(-gate), and voting-k-of-n gates, respectively. ■

Definition 4.2 (DFT leaf-types)**.** The set Leafs of *leaf-types* is defined as

$$\mathsf{Leafs} = \{\text{BE}, \text{CONST}(\top), \text{CONST}(\bot)\}$$

. Leafs of such types are called basic events, given-failure element, and fail-safe elements, respectively. ■

The set $\mathsf{Gates} \cup \mathsf{Leafs}$ of *element-types* is the union of gate-types and leaf-types. We extend the DFT with por-gates in Section 4.2.9 on page 98.

DFTs describe the behaviour of the system given the failures of components. For the stochastic analysis we feature in this thesis, the active and passive failure distributions are of interest.

Definition 4.3 (Component failures)**.** A *component failure* ω is a triple $\omega = (\mathsf{id}, \text{FD}_a, \text{FD}_p) \in \mathbb{N} \times \text{Distr}\mathbb{R}_{\geq 0} \times (\text{Distr}\mathbb{R}_{\geq 0} \cup \{\mathbf{0}\})$, with id a unique *identifier*, FD_a the *active failure distribution*, and FD_p the *passive failure distribution*. ■

Remark 18. The identifier is introduced for technical reasons, as a set of component failures would otherwise not allow the presence of multiple failures with identical failure distributions.

In this thesis, we assume the active failure distributions to be an exponential distribution with a rate $\lambda > 0$. Instead of writing $\text{FD} = \lambda e^{-\lambda x} \cdot u(x)$, we write $\mathcal{R}(\omega) = \lambda$. We assume the passive failure distribution either be an exponential distribution or the zero function. We therefore have for any $\omega = (i, \text{FD}_a, \text{FD}_p)$ that

$$\text{FD}_p = \alpha \times \text{FD}_a \text{ with } \alpha \in [0,1].$$

We call α the *dormancy factor* of ω denoted α_ω.

Definition 4.4 (Dynamic fault trees)**.** A DFT F^Ω over component failures Ω is a tuple $F^\Omega = (V, \sigma, \text{Tp}, \Theta, \text{top})$.

- V is a finite set of elements
- $\sigma\colon V \to V^*$ gives for each element v a word representing the *(ordered) successors* of v.
- a *type mapping* $\text{Tp}\colon V \to \mathsf{Gates} \cup \mathsf{Leafs}$, Tp injective. If $\text{Tp}(v) = K$, we may write $v \in K$.
- The attachment function $\Theta\colon \Omega \to \{v \in V \mid \text{Tp}(v) = \text{BE}\}$, s.t. Θ is injective.
- $\text{top} \in V$ is the *top-level element*. ■

We drop the superscript Ω whenever it is clear from the context. We often abuse the notation and write $\sigma(v)$ to denote the set

$$\{v' \in V \mid \exists i \in \mathbb{N} \text{ s.t. } \sigma(v)_i = v'\}.$$

The *degree of an element* v is defined as the out-degree of v, i.e. $\deg(v) = |\sigma(v)|$.

Definition 4.5 (Graph of a DFT)**.** Let $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$ be a DFT. We define $E(\sigma) = \{(v, v') \mid \exists v \in V.\, v' \in \sigma(v)\}$. Then $G_F = (V, E(\sigma))$ denotes the *graph of* F and $G_F^{\leftrightarrows} = (V, E(\sigma) \cup E(\sigma)^{-1})$ the *undirected graph of* F. ■

We define the predecessor set $\theta(v) = \{v' \mid v \in \sigma(v')\}$, and the closures as

$$\sigma^*(v) = \{v' \in V \mid \text{ there exists a path from } v \text{ to } v' \neq v \text{ in } G_F\}$$

and

$$\theta^*(v) = \{v' \in V \mid \text{ there exists a path from } v' \neq v \text{ to } v \text{ in } G_F\}.$$

Definition 4.6 (Element hierarchy)**.** Given a DFT $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$. The partial order R on V induced by reachability on $(V, E(\sigma))$ is the *element hierarchy* of F. Formally, $R \subseteq V \times V$ with $(v, v') \in R$ iff $v' \in \sigma^*(v)$. ■

For any DFT F with elements V and $K \in \mathsf{Gates} \cup \mathsf{Leafs}$, the set $F_K = \{v \in V \mid \text{Tp}(v) = K\}$ contains the nodes of F of type K. For any element from $v \in V$, we sometimes write that v is a K to denote that the type of v is K, i.e. $\text{Tp}(v) = K$.

Modules consist of sets of elements which are connected via *module paths*.

Definition 4.7 (module path)**.** Given a DFT $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$, a path $p = v_0 e_1 v_1 \dots e_n v_n$ through $G_F^{\leftrightarrows}$ is a *module path* if

- $\forall 0 \leq i \leq n\ v_i \notin \text{FDEP}$, and
- $\forall 0 \leq i < n\ v_i \in \text{SPARE} \implies v_{i+1} \notin \sigma(v_i)$.

Let $v, v' \in V$, we define the set $\text{spmp}_F(v, v')$ as $\{p = v_0 e_1 \dots e_n v_n \mid p\text{spare module path} \wedge v_0 = v \wedge v_n = v'\}$. ■

The set SMR_F of *spare module representatives of a DFT* F is $\mathsf{SMR}_F = \{v \in V \mid \exists s \in F_{\text{SPARE}}.\, v \in \sigma(s)\}$. The *extended module representatives of a DFT* F is $\mathsf{EMR}_F = \mathsf{SMR}_F \cup \{\top\}$. The *module* $\mathsf{EM}_{F,r}$ *represented by* r with $r \in \mathsf{EMR}_F$ is defined as

$$\mathsf{EM}_{F,r} = \{v \in V \mid \text{spmp}_F(v, v') \neq \emptyset\}$$

If $\mathsf{EM}_{F,r}$ is represented by a $r \neq \text{top}$, then $\mathsf{EM}_{F,r}$ is called a *spare module|see module*. We denote these modules with $\mathsf{SM}_{F,r}$. Otherwise, it is called the *top-module*. We drop the subscript F whenever it is clear from the context. For every spare-gate s, we call $\sigma(s)_1$ the *primary spare module representative* and the spare module it represents the *primary module*.

Remark 19. The notion of spare modules is more general than in Section 3.3.4.5 on page 46, as we do include elements connected to the module via undirected paths to be in the same module. This more general treatment eases modelling and simplifies some proofs.

Definition 4.8 (Module relation)**.** Given a DFT $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$. The module relation $\bowtie \subseteq V \times V$ is the smallest equivalence relation such that $(v, v') \in \bowtie$ if $\text{spmp}_F(v, v') \neq \emptyset$ ■

Basic events which have no attached component failure are called *dummy events*. For a dummy event e, we define $\mathcal{R}(e) = 0$. In the other cases, we often identify basic events with their attached component failure. For some basic event e with an attached component failure ω, $\Theta(\omega) = e$, We write $\mathcal{R}(e)$ for $\mathcal{R}(\omega)$ and α_e for α_ω.

Remark 20. Dummy events, unlike infallible elements, can be dependent events.

We recall from Chapter 3 that DFTs are acyclic. This, as well as other restrictions, is missing in the definition above.

Definition 4.9. We call a DFT $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$ well-formed if all of the following conditions hold.

1. The DFT is acyclic.

   $$G_F \text{ is acyclic}$$

2. Exactly the leaf types do not have successors.

   $$\forall v \in V\ \sigma(v) = \emptyset \iff \exists T \in \mathsf{Leafs} v \in T$$

3. The threshold of the voting gate is between one and the number of successors.

   $$\forall v \in \text{VOT}(k)\ 1 \leq k \leq |\sigma(v)|$$

4. The top level element is not an FDEP.

   $$\text{Tp}(\text{top}) \notin \text{FDEP}$$

5. FDEPs have no incoming edges.

   $$\forall v \in F_{\text{FDEP}}.\ \theta(v) = \emptyset$$

6. The second child of an FDEP is always a basic event.

   $$\forall v \in F_{\text{FDEP}}.\ \sigma(v)_2 \in F_{\text{BE}}$$

7. FDEPs have exactly two successors.

   $$\forall v \in F_{\text{FDEP}}.\ |\sigma(v)| = 2$$

8. Modules EM_r are independent.

   $$\forall \{r, r'\} \subseteq \mathsf{EMR}\ \mathsf{EM}_r \cap \mathsf{EM}_{r'} = \emptyset$$

9. Primary spare modules do not contain given-failure elements.

   $$\forall r \in \mathsf{SMR}\ \exists s \in F_{\text{SPARE}}\ r = \sigma(s)_1 \implies \mathsf{SM}_r \cap F_{\text{CONST}(\top)} = \emptyset.$$

10. Primary spare modules are never shared.

    $$\forall r \in \mathsf{SMR}\ \exists s \in F_{\text{SPARE}}\ r = \sigma(s)_1 \implies \forall s' \in F_{\text{SPARE}} \setminus \{s\}\ r \neq \sigma(s')_1$$ ■

The conditions (1-6) are standard. The condition that FDEPs have exactly two successors (7) simplifies further proofs. We introduce FDEPs with more successors as syntactic sugar in Section 4.2.9 on page 98. The independence of two spare modules (8) is a liberal restriction which prevents ambiguity in the meaning of claiming spares. Primary spare modules (9) are not allowed to contain given-failure elements s.t. we can simplify the definition of the initially claimed elements. This rather strict restriction can be removed in many cases, see Section 4.5 on page 110. Condition (10) is again a simplification. Primary modules are initially claimed by a unique predecessor. Therefore, they cannot be claimed later on and sharing them is superfluous.

In the case studies, we saw some examples where primary modules where shared with the top-module. This is not possible with the semantics described here. We refer to Section 4.5 on page 110 for some further treatment.

Assumption 3. *In the remainder of this chapter, we assume a DFT to be well-formed.*

4.2.2. Failure and event traces

We consider strictly ordered sequences of component failures, where each component failure occurs at most once.

To formalise, we consider a sequence ξ over a set K as an injective function from $\{1, \dots, n\}$ to K with $n \leq |K|$ to K. We call n the *length* of ξ, denoted by $|\xi|$. The set of all such sequences is denoted $K^{\triangleright}$. We often consider these sequences to be words over the alphabet K. We use ξ_i to denote the i'th item on the sequence ξ, i.e. $\xi(i)$. The last item of a sequence is denoted $\xi_{\downarrow} = \xi_{|\xi|}$. Furthermore, let $x \in K$. We write $x \in \xi$ if there exists $i \leq |\xi|$ s.t. $\xi_i = x$.

We write ε for the empty sequence, i.e. for the sequence mapping the empty set to K. Given a sequence $\xi = \xi_1 \dots \xi_n$ we define for each $m \leq n$ the prefix of length m as $\xi_{|m} = \xi_1 \dots \xi_m$. We abuse the notation and write $\xi_{|-i}$ to denote $\xi_{|(|\xi|-i)}$ for any $i > 0$. With $\mathrm{pre}(\xi) = \{\xi_{|i} \mid i < |\xi|\}$ we denote the set of all prefixes of ξ. We write $\xi \cdot x$ with $x \notin \xi$ for concatenation, i.e. given $\xi \colon \{1, \dots n\} \to K$, $\xi \cdot x : \{1, \dots, n, n+1\} \to K$, such that $\xi \cdot x(n+1) = x$ and for all $i \leq i$, $\xi \cdot x(i) = \xi(i)$.

Definition 4.10 ((Component) Failure trace)**.** Given a set of component failures Ω, we call any $\rho \in \Omega^{\triangleright}$ a *(component) failure trace.* ■

Definition 4.11 ((Basic) Event trace)**.** Given a DFT F, we call any $\pi^F \in F_{\mathrm{BE}}^{\triangleright}$ a *(basic) event trace over* F. ■

4.2.3. Introducing the running examples

We give two running examples, which are not motivated by reality, but by their compactness.

Remark 21. In the DFT, we display the attached component failure in the basic event. Often, we are not interested in the component failures and omit them.

The first example is a regularly used pattern.

Example 4.1. The DFT, depicted in Figure 4.1, describes a system of two components (radio1, radio2), which share a spare module (C). The system fails if both components fail. The DFT for this system is given by $F^{\Omega} = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ with $\Omega = \{a, b, c\}$ and

- $\{\mathrm{sys}, X, Y, A, B, C\}$
- σ given by
  - $\sigma(\mathrm{sys}) = XY$,
  - $\sigma(X) = AC$,
  - $\sigma(Y) = BC$,
  - $\sigma(A) = \sigma(B) = \sigma(C) = \varepsilon$.
- Tp given by $\mathrm{Tp}(\mathrm{sys}) = \mathrm{AND}$, $\mathrm{Tp}(X) = \mathrm{Tp}(Y) = \mathrm{SPARE}$, $\mathrm{Tp}(A) = \mathrm{Tp}(B) = \mathrm{Tp}(C) = \mathrm{BE}$.
- Θ given by $\Theta(a) = A$, $\Theta(b) = B$, $\Theta(c) = C$.
- $\mathrm{top} = \mathrm{sys}$ ▲.

The second example shows some functional dependencies in a very compact manner.

Example 4.2. The DFT, depicted in Figure 4.2, describes a system of three components (a, b, c). The failure of a causes b and c to fail, whereas the failure of b causes c to fail. The system fails if first a, then b and then c fails. The DFT for this system is given by $F^{\Omega} = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ with $\Omega = \{a, b, c\}$ and

- $\{\mathrm{sys}, X, Y, Z, A, B, C\}$
- σ given by
  - $\sigma(\mathrm{sys}) = ABC$,
  - $\sigma(X) = AB$,
  - $\sigma(Y) = AC$,
  - $\sigma(Z) = BC$,
  - $\sigma(A) = \sigma(B) = \sigma(C) = \varepsilon$.
- Tp given by $\mathrm{Tp}(\mathrm{sys}) = \mathrm{PAND}$, $\mathrm{Tp}(X) = \mathrm{Tp}(Y) = \mathrm{Tp}(Z) = \mathrm{FDEP}$, $\mathrm{Tp}(A) = \mathrm{Tp}(B) = \mathrm{Tp}(C) = \mathrm{BE}$.
- Θ given by $\Theta(a) = A$, $\Theta(b) = B$, $\Theta(c) = C$.
- $\mathrm{top} = \mathrm{sys}$ ▲.

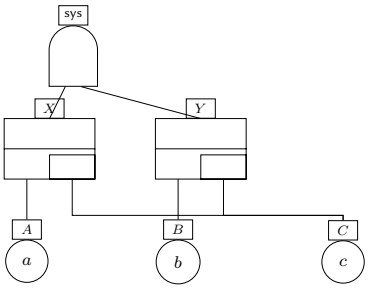


Figure 4.1.: The DFT from Example 4.1.

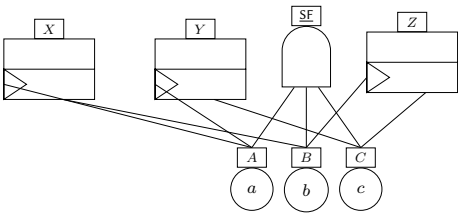


Figure 4.2.: The DFT from Example 4.2.

4.2.4. State of a DFT

In this section, we set out to formally describe the effect of a failed basic event to the internal state DFT. Therefore, we first have to give a definition of the state of a DFT. Notice that we ignore functional dependencies in this section and only discuss them later, when we discuss the effect of a component failure to the internal state of a DFT in Section 4.2.5 on page 91.

Intuition Our foremost interest is to define the set of event traces after which the top-level element fails. We use a structural induction to define the set of failed elements in the tree for a given event trace π. For static elements, we give an expression which encodes whether the element has failed, depending solely on the failure of the successor elements given π. For the pand-gate, failure combination also depends on the failure of the successor elements after prefixes of π. In a second step, we compress this information to a flag whether, up to the largest prefix of π, the successors have failed in the expected order. Spare gates claim a new element whenever their claimed successor fails. If no other successor can be claimed, the spare gate fails as well. Thus, the failure of spare gates depends on the information which representatives are claimed by which spare gate after the occurrence of π. From this information, we can extract the set of representatives which are available for a certain spare gate to claim.

We require that claiming cannot be undone, i.e., once claimed by a gate, a representative remains claimed by this gate. We assume that spare module representatives can only be claimed if they were not already claimed by some other gate before the basic event in π was triggered. Together with the independence of spare modules and the fact that we ignore functional dependencies here, resolving the interdependence between claiming and failing is simplified, as the semantics make sure that after each failure of a basic event, at most one spare gate claims another element[1]. This also ensures that no spare module representative is ever claimed by two different spare gates.

We describe the internal state of a DFT for each trace of basic events. Put it differently, we define the state of the DFT F after the subsequent occurrence of the events in the event trace π by using

[1] we prove this in Theorem 4.30.

two predicates, Failed(π) and ClaimedBy(π). The predicates describe which gates are considered failed after π occurred, and which spare module representatives are used, and if so, by which spare gate, respectively. We define them inductively over the length of π and (structure) inductively over the graph of the DFT.

Formal specification We introduce a construction of the mappings Failed and ClaimedBy, which we will characterise with the help of auxiliaries later in this section to ease both understanding and working with these definitions.

Remark 22. The construction we use here follows the definition and characterisation of the model-relation for CTL-formulae on labelled transition systems as given in [BK08].

Definition 4.12. Given an DFT $F^{\Omega} = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$. We define a relation $\models_F \subseteq V \times F_{\mathrm{BE}}^{\triangleright}$, called the *model-relation*, and a set $\{\dagger_F^s \subseteq V \times F_{\mathrm{BE}}^{\triangleright} \mid s \in F_{\mathrm{SPARE}}\}$, with a *claiming relation* for each spare gate.

The model relation and the claiming-relations are then defined as the smallest[1] relations such that for arbitrary $v \in V$ and $\pi \in F_{\mathrm{BE}}^{\triangleright}$:

- For $\pi = \varepsilon$, either:

  $v \in F_{\mathrm{BE}} \cup F_{\mathrm{CONST}}(\bot) \cup F_{\mathrm{SPARE}}$: The node does not fail.

  $$v \not\models_F \pi$$

  $v \in F_{\mathrm{CONST}}(\top)$: The node does fail.

  $$v \models_F \pi$$

  $v \in F_{\mathrm{AND}} \cup F_{\mathrm{PAND}}$: The node fails if all children fail:

  $$v \models_F \pi \iff \forall v' \in \sigma(v).\ v' \models_F \pi$$

  $v \in F_{\mathrm{OR}}$: The node fails if it has a child that has failed.

  $$v \models_F \pi \iff \exists v' \in \sigma(v).\ v' \models_F \pi$$

  $v \in F_{\mathrm{VOT}(k)}$: The node fails if at least k children have failed.

  $$v \models_F \pi \iff \exists \{v_1, \ldots, v_k\} \subseteq \sigma(v).\ \bigwedge_{i=1}^{k} v_i \models_F \pi$$

  and for any $s \in F_{\mathrm{SPARE}}$ and $v \in V$

  $$v \dagger_F^s \pi \iff v = \sigma(v)_1.$$

- For $|\pi| > 0$:

  For $\models$, the conditions of $\pi = \emptyset$ apply, unless defined below.

  $v \in F_{\mathrm{BE}}$: The node fails if the event has occurred before.

  $$v \models_F \pi \iff \exists j \text{ s.t. } \pi_j = v$$

  $v \in F_{\mathrm{PAND}}$: The node fail if all children fail and do it according their order.

  $$v \models_F \pi \iff \forall v' \in \sigma(v).\ v' \models_F \pi \wedge$$
  $$\forall \pi' \in \mathrm{pre}(\pi) \forall i < \deg(v).\ \sigma(v)_{i+1} \models_F \pi' \implies \sigma(v)_i \models \pi'$$

[1] indeed, the description is unique

$v \in F_{\text{SPARE}}$: The spare fails if all claimed children have failed and it cannot claim another.

$$v \models_F \pi \iff \forall v' \in \sigma(v)\,(v' \dagger_F^v \pi_{|-1} \implies v' \models_F \pi)$$
$$v' \not\dagger_F^v \pi_{|-1} \implies (v' \models_F \pi \vee \exists s \in F_{\text{SPARE}}\; v' \dagger_F^s \pi_{|-1})$$

Let $s \in F_{\text{SPARE}}$, if

$$s \not\models \pi \wedge \forall v' \in \sigma(s).\; v \dagger_F^s \pi_{|-1} \implies v \models_F \pi$$

then we call s *claiming after* π. For each $s \in F_{\text{SPARE}}$, we define the *next spare for* s *at* π as

$$\text{NextSpare}(\pi,s) = \min\{\sigma(v)_j \in V \mid \sigma(v)_j \not\models_F \pi \wedge\; \nexists s' \in F_{\text{SPARE}}\; \sigma(v)_j \dagger_F^{s'} \pi_{|-1}\}.$$

Now for all $v \in V$ and $s \in F_{\text{SPARE}}$:

$$v \dagger_F^s \pi \iff s \text{ claiming and } v = \text{NextSpare}(\pi,s) \vee v \dagger_F^s \pi_{|-1}$$ ■

Remark 23. The definitions of the model relation for basic events and priority-and gates in the inductive step (i.e. $|\pi| > 0$) are also correct for the base case $\pi = \varepsilon$.

The semantics allow a SMR to be claimed by a set of spares. In Corollary 4.31 on page 90, we show that the cardinality of the set is always less or equal 1. Moreover, we show that the primary component of a spare never fails at $\pi = \varepsilon$ in Corollary 4.17 on page 87.

Instead of the relations introduced above, we use the following mappings in the remainder.

Definition 4.13. Let $F^\Omega = (V, \sigma, \text{Tp}, \Theta, \text{top})$ be a DFT. We define the *set of failed elements after* π as

$$\text{Failed}\colon F_{\text{BE}}^{\triangleright} \to \mathcal{P}(V),$$
$$\text{Failed}(\pi) = \{v \in V \mid v \models_F \pi\},$$

and the mapping of *representatives claimed by after* π as

$$\text{ClaimedBy}\colon F_{\text{BE}}^{\triangleright} \to (V \to \mathcal{P}(F_{\text{SPARE}}))$$
$$\text{ClaimedBy}(\pi)(v) = \{s \in F_{\text{SPARE}} \mid v \dagger_F^s \pi\}$$ ■

We define auxiliary mappings to which simplify notation and then give an characterisation of the mappings above.

The following map describes which elements are available for claiming given a spare-gate.

$$\text{Available}\colon F_{\text{BE}}^{\triangleright} \setminus \{\varepsilon\} \to F_{\text{SPARE}} \to \mathcal{P}(V)$$
$$\text{Available}(\pi)(v) = \{v' \in \sigma(v) \mid \text{ClaimedBy}(\pi_{|-1})(v') = \emptyset \wedge v' \notin \text{Failed}(\pi)\}$$

We defined being claimed by from the perspective of a SMR. We the following map gives is from the viewpoint of a spare-gate.

$$\text{Claimed}\colon F_{\text{BE}}^{\triangleright} \to (F_{\text{SPARE}} \to \mathcal{P}(V))$$
$$\text{Claimed}(\pi)(v) = \{v' \in \sigma(v) \mid v \in \text{ClaimedBy}(\pi)(v')\}$$

Furthermore, we use the following definition to denote that some element v failed before or simultaneous with v.

$$\text{FB}_\pi(v, v') = \forall \pi' \in \text{pre}(\pi).\; v' \in \text{Failed}(\pi') \implies v \in \text{Failed}(\pi')$$

The following lemmas characterise these sets.

Lemma 4.1 (Characterisation of Failed(π))**.** *Given a DFT* $F^\Omega = (V, \sigma, Tp, \Theta, top)$ *and* $\pi \in F_{BE}^{\triangleright}$*. We characterise Failed*(π) *via an induction over the length of* π*:*

$\pi = \varepsilon$: *We use a induction over the structure of* (V, E).

$v \in F_{BE} \cup F_{SPARE} \cup F_{CONST(\bot)}$: *The node is not in the set of failed nodes.*

$$v \notin \mathit{Failed}(\pi)$$

$v \in F_{CONST(\top)}$: *The node is in the set of failed nodes.*

$$v \in \mathit{Failed}(\pi)$$

$v \in F_{AND}$: *The node has failed iff all children have failed:*

$$v \in \mathit{Failed}(\pi) \iff \sigma(v) \subseteq \mathit{Failed}(\pi)$$

$v \in F_{OR}$: *The node has failed iff it has a child that has failed.*

$$v \in \mathit{Failed}(\pi) \iff \sigma(v) \cap \mathit{Failed}(\pi) \neq \emptyset$$

$v \in F_{VOT(k)}$: *The node failed if at least* k *children have failed.*

$$v \in \mathit{Failed}(\pi) \iff |\sigma(v) \cap \mathit{Failed}(\pi)| \geq k$$

$v \in F_{PAND}$: *The node failed if all children have failed according to the given ordering.*

$$v \in \mathit{Failed}(\pi) \iff \sigma(v) \subseteq \mathit{Failed}(\pi) \ \wedge \forall i < \deg(v) \mathit{FB}_\pi(\sigma(v)_{i+1}, \sigma(v)_i)$$

$|\pi| > 0$: *We use a induction over the structure of* (V, E). *The characterisations of* $\pi = \varepsilon$ *apply, unless defined below.*

$v \in F_{BE}$: *The node fails if the event has occurred before.*

$$v \in \mathit{Failed}(\pi) \iff \exists j \leq |\pi|.\ \pi_j = v$$

$v \in F_{SPARE}$: *The spare fails if all claimed children have failed and it cannot claim another.*

$$v \in \mathit{Failed}(\pi) \iff \mathit{Claimed}(\pi_{|-1})(v) \subseteq \mathit{Failed}(\pi) \wedge \mathit{Available}(\pi)(v) = \emptyset$$

Proof. We distinguish the different cases as in the lemma.

$v \in F_{\text{AND}}$:

$$\begin{aligned} v \in \text{Failed}(\pi) &\iff v \models_F \pi \\ &\iff \forall v' \in \sigma(v)\ v' \models_F \pi \\ &\iff \forall v' \in \sigma(v)\ v' \in \text{Failed}(\pi) \iff \sigma(v) \subseteq \text{Failed}(\pi) \end{aligned}$$

$v \in F_{\text{OR}}$:

$$\begin{aligned} v \in \text{Failed}(\pi) &\iff v \models_F \pi \\ &\iff \exists v' \in \sigma(v)\ v' \models_F \pi \\ &\iff \exists v' \in \sigma(v)\ v' \in \text{Failed}(\pi) \iff \sigma(v) \cap \text{Failed}(\pi) \neq \emptyset \end{aligned}$$

$v \in F_{\text{VOT}(k)}$:

$$
\begin{aligned}
v \in \text{Failed}(\pi) &\iff v \models_F \pi \\
&\iff \exists \{v_1, \ldots, v_k\} \subseteq \sigma(v) \ \bigwedge_{j=1}^{k} v_j \models_F \pi \\
&\iff \exists \{v_1, \ldots, v_k\} \subseteq \sigma(v) \ \bigwedge_{j=1}^{k} v_j \in \text{Failed}(\pi) \\
&\iff \exists \{v_1, \ldots, v_k\} \subseteq \sigma(v) \ \{v_1, \ldots, v_k\} \subseteq \text{Failed}(\pi) \\
&\iff |\sigma(v) \cap \text{Failed}(\pi)| \geq k
\end{aligned}
$$

$v \in F_{\text{CONST}(\bot)}$ or $v \in F_{\text{BE}} \cup F_{\text{SPARE}}$ with $\pi = \varepsilon$:

$$v \notin \text{Failed}(\pi) \iff v \not\models_F \pi$$

$v \in F_{\text{CONST}(\bot)}$ or $v \in F_{\text{BE}} \cup F_{\text{SPARE}}$ with $\pi = \varepsilon$:

$$v \in \text{Failed}(\pi) \iff v \models_F \pi$$

$v \in F_{\text{BE}}$, $\pi \neq \varepsilon$:

$$
\begin{aligned}
v \in \text{Failed}(\pi) &\iff v \models_F \pi \\
&\iff \exists j \leq |\pi|.\ \pi_j = v
\end{aligned}
$$

$v \in F_{\text{SPARE}}$, $\pi \neq \varepsilon$:

$$
\begin{aligned}
& v \in \text{Failed}(\pi) \iff v \models_F \pi && \iff \\
& \forall v' \in \sigma(v)(v' \dagger_F^v \pi_{|-1} \implies v' \models_F \pi) \wedge \\
& (v' \dagger_F^v \pi_{|-1} \implies v' \models_F \pi \vee \exists s \in F_{\text{SPARE}}\ v' \dagger_F^s \pi_{|-1}) && \iff \\
& \{v' \in \sigma(v) \mid v' \dagger_F^v \pi_{|-1}\} \subseteq \{v' \in \sigma(v) \mid v' \models_F \pi)\} \wedge \\
& \{v' \in \sigma(v) \mid v' \dagger_F^v \pi_{|-1}\} \subseteq \{v' \in \sigma(v) \mid v' \models_F \pi \vee \exists s \in F_{\text{SPARE}} v' \dagger_F^s \pi_{|-1})\} && \iff \\
& \text{Claimed}(\pi_{|-1}) \subseteq \text{Failed}(\pi) \wedge \\
& \{v' \in \sigma(v) \mid v' \dagger_F^v \pi_{|-1}\} \subseteq \{v' \in \sigma(v) \mid v' \models_F \pi \vee \exists s \in F_{\text{SPARE}} v' \dagger_F^s \pi_{|-1})\} && \iff \\
& \text{Claimed}(\pi_{|-1}) \subseteq \text{Failed}(\pi) \wedge \\
& \sigma(v) \subseteq \{v' \in \sigma(v) \mid v' \models_F \pi \vee \exists s \in F_{\text{SPARE}} v' \dagger_F^s \pi_{|-1})\} && \iff \\
& \text{Claimed}(\pi_{|-1}) \subseteq \text{Failed}(\pi) \wedge \\
& \sigma(v) \cap \{v' \in \sigma(v) \mid \neg v' \models_F \pi \wedge \forall s \in F_{\text{SPARE}} \neg v' \dagger_F^s \pi_{|-1})\} = \emptyset && \iff \\
& \text{Claimed}(\pi_{|-1}) \subseteq \text{Failed}(\pi) \wedge \\
& \{v' \in \sigma(v) \mid v' \notin \text{Failed}(\pi) \wedge \text{ClaimedBy}(\pi)(v') = \emptyset\} = \emptyset && \iff \\
& \text{Claimed}(\pi_{|-1}) \subseteq \text{Failed}(\pi) \wedge \text{Available}(\pi)(v) = \emptyset
\end{aligned}
$$

$v \in F_{\text{PAND}}$:

$$
\begin{aligned}
& v \in \text{Failed}(\pi) \iff v \models_F \pi && \iff \\
& \forall v' \in \sigma(v).\ v' \models_F \pi \wedge \\
& (\pi = \varepsilon \vee \forall \pi' \in \text{pre}(\pi).\ \forall i < \deg(v).\ \sigma(v)_{i+1} \models_F \pi' \implies \sigma(v)_i \models_F \pi') && \iff \\
& \forall v' \in \sigma(v).\ v' \models_F \pi \wedge \\
& \forall \pi' \in \text{pre}(\pi).\ \forall i < \deg(v).\ \sigma(v)_{i+1} \models \pi' \implies \sigma(v)_i \models_F \pi' && \iff \\
& \sigma(v) \subseteq \text{Failed}(\pi) \wedge \\
& \forall i < \deg(v).\ \forall \pi' \in \text{pre}(\pi).\ \sigma(v)_{i+1} \in \text{Failed}(\pi') \implies \sigma(v)_i \in \text{Failed}(\pi') && \iff \\
& \sigma(v) \subseteq \text{Failed}(\pi) \wedge \forall i < \deg(v).\ \text{FB}(\sigma(v)_{i+1}, \sigma(v)_i)
\end{aligned}
$$

□

Lemma 4.2 (Characterisation of ClaimedBy(ε))**.** *Let $F = (V, \sigma, Tp, \Theta, top)$ be a DFT. The following statement holds.*

$$\forall v \in \mathsf{SMR} \forall s \in F_{SPARE}.\ v = \sigma(s)_1 \implies s \in ClaimedBy(\varepsilon, v)$$

Proof. From Definition 4.12 we know

$$v \dagger_F^s \varepsilon \iff v = \sigma(v)_1. \qquad \square$$

The following corollary is an equivalent statement from the perspective of an spare-gate.

Corollary 4.3. *Let F be a DFT. The following statement holds.*

$$\forall s \in F_{SPARE}.\ Claimed(\varepsilon, s) = \{\sigma(s)_1\}.$$

The following statement is a direct corollary to Definition 4.12.

Corollary 4.4. *Let F be a DFT and $\pi \in F_{BE}^{\triangleright}, x \in F_{BE}$ and $v \in \mathsf{SMR}_F$. It holds that*

$$\begin{aligned} ClaimedBy(\pi \cdot x, v) \quad =& ClaimedBy(\pi, v) \cup \\ & \{s \in F_{SPARE} \mid v = NextSpare(\pi \cdot x, v) \text{ and } s \text{ claiming after } \pi \cdot x\} \end{aligned}$$

Proposition 4.5. *Let F be a DFT and $\pi \in F_{BE}^{\triangleright}, x \in F_{BE}$ and $s \in F_{SPARE}$. It holds that*

$$Claimed(\pi \cdot x, s) = Claimed(\pi, s) \cup \begin{cases} \{NextSpare(\pi \cdot x, v)\} & \text{if } s \text{ claiming after } \pi \cdot x \\ \emptyset & \text{else.} \end{cases}$$

Proof.

$$\begin{aligned} &\text{Claimed}(\pi \cdot x, s) && = \\ &\{v' \in \sigma(s) \mid s \in \text{ClaimedBy}(\pi \cdot x, v')\} && = \\ &\{v' \in \sigma(s) \mid v' = \text{NextSpare}(\pi \cdot x, s) \text{ and } s \text{ claiming after } \pi \cdot x \\ &\quad \vee s \in \text{ClaimedBy}(\pi, v')\} && = \\ &\{\text{NextSpare}(\pi \cdot x, s) \mid \text{ and } s \text{ claiming after } \pi \cdot x\} \cup \\ &\quad \{v' \in \sigma(s) \mid s \in \text{ClaimedBy}(\pi, v')\} && = \\ &\text{Claimed}(\pi, s) \cup \{\text{NextSpare}(\pi \cdot x, s) \mid \text{ and } s \text{ claiming after } \pi \cdot x\} && \square \end{aligned}$$

The following characterisation is a directly rewritten representation of NextSpare.

Corollary 4.6. *Let F be a DFT and $\pi \in F_{BE}^{\triangleright}$ and $s \in F_{SPARE}$. $NextSpare(\pi, s) = \min_j \{\sigma(v)_j \in V \mid \sigma(v)_j \notin Failed(\pi) \wedge ClaimedBy(\pi_{|-1}, \sigma(v)_j) = \emptyset\}$.*

It remains to show our claims that no SMR is never claimed by more than one spare. Furthermore, we want to formalise some properties of failure propagation and claiming, to show the accuracy of the presented semantics and to streamline further arguments.

Properties of DFTs and their behaviour The presented proofs are rather technical, but also give a nice hands-on introduction. We start with the fact that claiming is never undone and that claiming is only done by spare gates which are operational.

Lemma 4.7. *Let F be a DFT with elements V. Let $\pi, \pi' \in F_{BE}^{\triangleright}$ s.t. $\pi' \in pre(\pi)$. It holds that for all $s \in F_{SPARE}$,*

$$Claimed(\pi', s) \subseteq Claimed(\pi, s)$$

and that

$$\{v \in V \mid ClaimedBy(\pi', v) = \emptyset\} \supseteq \{v \in V \mid ClaimedBy(\pi, v) = \emptyset\}.$$

Proof. We show the claims for $\pi = \pi' \cdot x$. For the first point, we simply deduce

$$\text{Claimed}(\pi' \cdot x, s) = \text{Claimed}(\pi') \cup X \text{for some } X \implies \text{Claimed}(\pi' \cdot x) \supseteq \text{Claimed}(\pi', s')$$

For the second point,

$$\begin{array}{lll}
\{v \in V & \mid \text{ClaimedBy}(\pi', v) = \emptyset\} \supseteq \{v \in V \mid \text{ClaimedBy}(\pi' \cdot x, v) = \emptyset\} & \implies \\
\{v \in V & \mid \text{ClaimedBy}(\pi', v) \neq \emptyset\} \cap \{v \in V \mid \text{ClaimedBy}(\pi' \cdot x, v) = \emptyset\} = \emptyset & \implies \\
\{v \in V & \mid \text{ClaimedBy}(\pi', v) \neq \emptyset \wedge \text{ClaimedBy}(\pi' \cdot x, v) = \emptyset\} = \emptyset & \implies \\
\{v \in V & \mid \exists s \in F_{\text{SPARE}}.\ s \in \text{ClaimedBy}(\pi', v) \wedge & \\
 & \quad \forall s \in F_{\text{SPARE}}.\ s \notin \text{ClaimedBy}(\pi' \cdot x, v)\} = \emptyset & \implies \\
\{v \in V & \mid \exists s \in F_{\text{SPARE}}.\ s \in \text{ClaimedBy}(\pi', v) \wedge s \notin \text{ClaimedBy}(\pi' \cdot x, v)\} = \emptyset & \implies \\
\forall v \in V. & \neg(\exists s \in F_{\text{SPARE}}.\ s \in \text{ClaimedBy}(\pi', v) \wedge s \notin \text{ClaimedBy}(\pi' \cdot x, v)) & \implies \\
\forall v \in V. & \forall s \in F_{\text{SPARE}}.\ \neg s \in \text{ClaimedBy}(\pi', v) \vee s \in \text{ClaimedBy}(\pi' \cdot x, v) & \implies \\
\forall v \in V. & \forall s \in F_{\text{SPARE}}.\ (s \in \text{ClaimedBy}(\pi', v) \implies s \in \text{ClaimedBy}(\pi' \cdot x, v)) & \implies \\
\forall v \in V. & \text{ClaimedBy}(\pi, v) \subseteq \text{ClaimedBy}(\pi \cdot x, v) &
\end{array}$$

which follows as for the first point. The lemma now follows with induction over the length π' and transitivity of $\subseteq$. □

Proposition 4.8. *Let F be a DFT with $s \in F_{SPARE}$ and $\pi \in F_{BE}^{\triangleright}$.*

$$s \in \textit{Failed}(\pi) \implies \textit{Claimed}(\pi_{|-1}, v) = \textit{Claimed}(\pi, v)$$

Proof. By Lemma 4.7, it suffices to show

$$s \in \text{Failed}(\pi) \implies \text{Claimed}(\pi_{|-1}, s) \supseteq \text{Claimed}(\pi, s)$$

To the contrary, let us assume $\text{Claimed}(\pi, s) \not\subseteq \text{Claimed}(\pi_{|-1}, s)$.

$$\begin{array}{ll}
\neg(\text{Claimed}(\pi, s) \subseteq \text{Claimed}(\pi_{|-1}, s)) & \implies \\
\neg(\{v \in \sigma(s) \mid s \in \text{ClaimedBy}(\pi, v)\} \subseteq \{v \in \sigma(s) \mid s \in \text{ClaimedBy}(\pi_{|-1}, v)\}) & \implies \\
\{v \in \sigma(s) \mid s \notin \text{ClaimedBy}(\pi, v)\} \cap \{v \in \sigma(s) \mid s \in \text{ClaimedBy}(\pi_{|-1}, v)\} \neq \emptyset & \implies \\
\{v \in \sigma(s) \mid s \notin \text{ClaimedBy}(\pi, v) \wedge s \in \text{ClaimedBy}(\pi_{|-1}, v)\} \neq \emptyset & \implies \\
\exists v \in \sigma(s) s \notin \text{ClaimedBy}(\pi, v) \wedge s \in \text{ClaimedBy}(\pi_{|-1}, v) & \implies \\
\exists v \in \sigma(s) \neg v \dagger_F^s \pi \wedge \gamma_{v,s}^{\pi_{|-1}} & \implies \\
s \text{ claiming and } v = \text{NextSpare}(\pi, s) & \implies \\
s \not\models_F \pi \wedge \forall v' \in \sigma(s)(v' \dagger_F^s \pi_{|-1} \implies v' \models_F \pi) & \implies \\
s \notin \text{Failed}(\pi) &
\end{array}$$

Which is a contradiction to the premise, so our assumption must be wrong. □

Next, we show that indeed, an element never recovers from a failure. We refer to this property as *coherency*. We need to show simultaneously that once an element is unavailable for claiming by a spare gate, it remains unavailable.

Proposition 4.9. *Let F be a DFT. Let $\pi, \pi' \in F_{BE}^{\triangleright}$ and $\pi' \in pre(\pi)$. It holds that*

- *Failed*$(\pi') \subset$ *Failed*(π)*, and*
- $\forall v \in F_{SPARE}$. *Available*$(\pi', v) \supseteq$ *Available*(π, v).

Proof. We prove $\text{Failed}(\pi') \subseteq \text{Failed}(\pi)$ and $\forall v \in F_{\text{SPARE}}.\ \text{Available}(\pi', v) \supseteq \text{Available}(\pi, v)$ by a structural induction over the graph of F, showing that

$$v \in \text{Failed}(\pi') \implies v \in \text{Failed}(\pi)$$

by using

$$\forall \sigma(v).\ \sigma(v) \in \text{Failed}(\pi') \implies \sigma(v) \in \text{Failed}(\pi').$$

Based on this assumption, for every $v \in V$:

$$\begin{aligned}\text{Available}(\pi', v) &= \{v' \in \sigma(v) \mid \text{ClaimedBy}(\pi'_{-1}) = \emptyset \wedge v' \notin \text{Failed}(\pi')\} \\ &\supseteq \{v' \in \sigma(v) \mid \text{ClaimedBy}(\pi'_{-1}) = \emptyset \wedge v' \notin \text{Failed}(\pi)\} \\ &\supseteq \{v' \in \sigma(v) \mid \text{ClaimedBy}(\pi_{-1}) = \emptyset \wedge v' \notin \text{Failed}(\pi)\} \\ &= \text{Available}(\pi, v)\end{aligned}$$

For $\text{Failed}(\pi)$, we make a case distinction.

$v \in F_{\text{BE}}$:

$$v \in \text{Failed}(\pi') \implies v \in \pi' \implies v \in \pi \implies v \in \text{Failed}(\pi')$$

$v \in F_{\text{AND}}$:

$$v \in \text{Failed}(\pi') \implies \sigma(v) \subseteq \text{Failed}(\pi') \implies \sigma(v) \subseteq \text{Failed}(\pi) \implies v \in \text{Failed}(\pi)$$

$v \in F_{\text{OR}}$:

$$v \in \text{Failed}(\pi') \implies \sigma(v) \cap \text{Failed}(\pi') \neq \emptyset \implies \exists v' \in \sigma(v).\ v' \in \text{Failed}(\pi') \implies$$
$$\exists v' \in \sigma(v).\ v' \in \text{Failed}(\pi) \implies \sigma(v) \cap \text{Failed}(\pi) \neq \emptyset \implies v \in \text{Failed}(\pi)$$

$v \in F_{\text{VOT}(k)}$: Using the same expansion as for OR, we get

$$\begin{aligned}v \in \text{Failed}(\pi') \implies \quad & |\sigma(v) \cap \text{Failed}(\pi')| \geq k \implies \ldots \implies \\ & |\sigma(v) \cap \text{Failed}(\pi)| \geq k \implies v \in \text{Failed}(\pi)\end{aligned}$$

$v \in F_{\text{PAND}}$:

$$\begin{aligned}& v \in \text{Failed}(\pi') && \implies \\ & \sigma(v) \subseteq \text{Failed}(\pi') \wedge \forall 1 \leq i < \deg(v).\ \text{FB}_{\pi'}(\sigma(v)_i, \sigma(v)_{i+1}) && \implies \\ & \sigma(v) \subseteq \text{Failed}(\pi') \wedge \\ & \quad \forall 1 \leq i < \deg(v)\ \forall \hat{\pi} \in \text{pre}(\pi').\ \sigma(v)_{i+1} \in \text{Failed}(\pi') \implies \sigma(v)_i \in \text{Failed}(\pi') && \implies \\ & \sigma(v) \subseteq \text{Failed}(\pi) \wedge\ \forall 1 \leq i < \deg(v) \\ & \quad \forall \hat{\pi} \in \text{pre}(\pi').\ \sigma(v)_{i+1} \in \text{Failed}(\hat{\pi}) \implies \sigma(v)_i \in \text{Failed}(\hat{\pi}) \wedge \\ & \quad \text{(By coherency, the right hand side is always true.)} \\ & \quad \forall \hat{\pi} \in \text{pre}(\pi) \setminus \text{pre}(\pi').\ \sigma(v)_{i+1} \in \text{Failed}(\hat{\pi}) \implies \sigma(v)_i \in \text{Failed}(\hat{\pi}) && \implies \\ & \sigma(v) \subseteq \text{Failed}(\pi) \wedge \\ & \quad \forall 1 \leq i < \deg(v)\ \forall \hat{\pi} \in \text{pre}(\pi).\ \sigma(v)_{i+1} \in \text{Failed}(\hat{\pi}) \implies \sigma(v)_i \in \text{Failed}(\hat{\pi}) && \implies \\ & v \in \text{Failed}(\pi)\end{aligned}$$

$v \in F_{\text{SPARE}}$: We show that $v \in \text{Failed}(\pi') \implies v \in \text{Failed}(\pi' \cdot x)$ for $x \in F_{\text{BE}}$. By induction over

the length of π', we can then conclude that $v \in \text{Failed}(\pi') \implies v \in \text{Failed}(\pi)$.

$$
\begin{aligned}
& v \in \text{Failed}(\pi') && \implies \\
& \text{Claimed}(\pi'_{|-1}, v) \subseteq \text{Failed}(\pi') \wedge \text{Available}(\pi', v) = \emptyset && \implies \\
& \text{Claimed}(\pi'_{|-1}, v) \subseteq \text{Failed}(\pi' \cdot x) \wedge \text{Available}(\pi', v) = \emptyset && \implies \\
& \text{(with Propostion 4.8)} \\
& \text{Claimed}(\pi', v) \subseteq \text{Failed}(\pi' \cdot x) \wedge \text{Available}(\pi', v) = \emptyset && \implies \\
& \text{Claimed}(\pi', v) \subseteq \text{Failed}(\pi' \cdot x) \wedge \text{Available}(\pi' \cdot x, v) = \emptyset && \implies \\
& v \in \text{Failed}(\pi \cdot x)
\end{aligned}
$$

It remains to show that $\text{Failed}(\pi') \neq \text{Failed}(\pi)$. Consider $\pi_\downarrow$, it holds that $\pi_\downarrow \notin \pi'$ and thus $\pi_\downarrow \notin \text{Failed}(\pi')$. □

We are often interested in analysing what the effect of a failure is, i.e. how it propagates. Therefore, we define a map which encodes which elements have just failed, i.e. $\text{JustFailed}(\pi)$ denotes those gates whose failure is triggered by the failure of $\pi_\downarrow$. Let $x \in F_{\text{BE}}$.

$$
\begin{aligned}
& \text{JustFailed}\colon F_{\text{BE}}^{\triangleright} \to \mathcal{P}(V) \\
& \text{JustFailed}(\varepsilon) \quad = \text{Failed}(\varepsilon) \\
& \text{JustFailed}(\pi \cdot x) = \text{Failed}(\pi \cdot x) \setminus \text{Failed}(\pi)
\end{aligned}
$$

Proposition 4.10. *Let F be a DFT and $v \in F_{AND} \cup F_{OR} \cup F_{VOT(k)}$. Then*

$$v \in \mathit{JustFailed}(\pi) \implies \exists v' \in \sigma(v).\ v' \in \mathit{JustFailed}(\pi).$$

We only show this for the voting gate[1].

Proof. Let $v \in F_{\text{VOT}(k)}$ with $k > 0$. We have that $\text{JustFailed}(\varepsilon) = \text{Failed}(\varepsilon)$, and thus $v \in \text{JustFailed}(\varepsilon) \implies |\sigma(v) \cap \text{Failed}(\varepsilon)| \geq k \geq 1$. Now consider $\pi = \pi' \cdot x$.

$$
\begin{aligned}
& v \in \text{JustFailed}(\pi' \cdot x) && \implies \\
& v \in \text{Failed}(\pi' \cdot x) \wedge v \notin \text{Failed}(\pi') && \implies \\
& |\sigma(v) \cap \text{Failed}(\pi' \cdot x)| \geq k \wedge |\sigma(v) \cap \text{Failed}(\pi')| < k && \implies \\
& |\sigma(v) \cap \text{Failed}(\pi' \cdot x)| - |\sigma(v) \cap \text{Failed}(\pi')| > 0 && \implies \\
& \text{(by coherency)} \\
& |\sigma(v) \cap \text{Failed}(\pi' \cdot x) \setminus \text{Failed}(\pi')| > 0 && \implies \\
& \exists v' \in \sigma(v).\ v' \in \text{Failed}(\pi' \cdot x) \wedge v' \notin \text{Failed}(\pi') && \implies \\
& \exists v' \in \sigma(v).\ v' \in \text{JustFailed}(\pi' \cdot x). && \square
\end{aligned}
$$

An alternative characterisation for PANDs allows simplified proofs. A pand-gate can only fail as long as its children failed according the given ordering, thus $\text{Failable}(\pi)$ denotes those gates which still can fail.

$$
\begin{aligned}
& \text{Failable}\colon F_{\text{BE}}^{\triangleright} \to \mathcal{P}(F_{\text{PAND}}) \\
& \text{Failable}(\pi) = \begin{cases} F_{\text{PAND}} & \pi = \varepsilon \\ \{v \in F_{\text{PAND}} \mid v \in \text{Failable}(\pi_{|-1}) \wedge \\ \quad (\forall i < \deg(v)\ \sigma(v)_{i+1} \in \text{JustFailed}(\pi) \implies \sigma(v)_i \in \text{Failed}(\pi))\} & \text{else.} \end{cases}
\end{aligned}
$$

With this predicate, we can formulate an equivalent characterisation of $\text{Failed}(\pi)$ for the case of pand-gates.

[1] As and-gates and or-gates are just special cases, the proof is formally complete.

Lemma 4.11 (PAND with Failable). *Let F be an DFT and $v \in F_{PAND}$, furthermore let $\pi \in F_{BE}^{\triangleright} \setminus \{\varepsilon\}$.*

$$v \in \mathit{Failed}(\pi) \iff \sigma(v) \subseteq \mathit{Failed}(\pi) \wedge \mathit{Failable}(\pi_{|-1})$$

Proof. $\Leftarrow$. We only have to show that

$$\sigma(v) \subseteq \text{Failed}(\pi) \ \wedge v \in \text{Failable}(\pi_{|-1}) \implies \forall 1 \leq i < \deg(v).\ \text{FB}_\pi(\sigma(v)_i, \sigma(v)_{i+1})$$

$$\begin{aligned}
&\sigma(v) \subseteq \text{Failed}(\pi) \ \wedge v \in \text{Failable}(\pi_{|-1}) && \implies \\
&\sigma(v) \subseteq \text{Failed}(\pi) \ \wedge v \in \text{Failable}(\pi_{|-2}) \wedge \\
&\forall i \in \{1, \ldots, \deg(v)\} \sigma(v)_{i+1} \in \text{JustFailed}(\pi_{|-1}) \implies \sigma(v)_i \in \text{Failed}(\pi_{|-1}) && \implies \\
&\ldots && \implies \\
&\forall \pi' \in \text{pre}(\pi) \forall i \in \{1, \ldots, \deg(v)\} \\
&\quad \sigma(v)_{i+1} \in \text{JustFailed}(\pi') \implies \sigma(v)_i \in \text{Failed}(\pi') && \implies \\
&\forall i \in \{1, \ldots, \deg(v)\}\ \forall \pi' \in \text{pre}(\pi) \\
&\quad \sigma(v)_{i+1} \in \text{Failed}(\pi') \wedge \sigma(v)_{i+1} \notin \text{Failed}(\pi'_{|-1}) \implies \sigma(v)_i \in \text{Failed}(\pi')
\end{aligned}$$

We partition the combinations of i and π' into the cases where $\sigma(v)_{i+1} \notin \text{Failed}(\pi'_{|-1})$ holds. The other case are those combinations of i and π' for which $\sigma(v)_{i+1} \notin \text{Failed}(\pi'_{|-1})$ does not hold. For those, it remains to show that

$$\sigma(v)_{i+1} \in \text{Failed}(\pi') \wedge \sigma(v)_{i+1} \in \text{Failed}(\pi'_{|-1}) \implies \sigma(v)_{i+1} \in \text{Failed}(\pi'),$$

which follows directly from coherency and the fact that $\text{Failed}(\varepsilon) = \emptyset$ as outlined below.

$$\begin{aligned}
&\sigma(v)_{i+1} \in \text{Failed}(\pi') \wedge \sigma(v)_{i+1} \in \text{Failed}(\pi'_{|-1}) && \implies \\
&\exists \hat{\pi} \in \text{pre}(\pi) \sigma(v)_{i+1} \in \text{JustFailed}(\hat{\pi}) && \implies \\
&\exists \hat{\pi} \in \text{pre}(\pi) \sigma(v) \in \text{Failed}(\hat{\pi}) && \implies \\
&\sigma(v) \in \text{Failed}(\pi')
\end{aligned}$$

Merging both cases we get

$$\forall i \in \{1, \ldots, \deg(v)\}\ \forall \pi' \in \text{pre}(\pi)\ \sigma(v)_{i+1} \in \text{Failed}(\pi') \implies \sigma(v)_i \in \text{Failed}(\pi')$$

which is by definition

$$\forall 1 \leq i < \deg(v).\ \text{FB}_\pi(\sigma(v)_i, \sigma(v)_{i+1})$$

$\Rightarrow$. We only have to show that

$$\sigma(v) \subseteq \text{Failed}(\pi) \wedge \forall 1 \leq i < \deg(v).\ \text{FB}_\pi(\sigma(v)_i, \sigma(v)_{i+1}) \implies v \in \text{Failable}(\pi_{|-1})$$

$$\begin{aligned}
&\forall 1 \leq i < \deg(v).\ \text{FB}_\pi(\sigma(v)_i, \sigma(v)_{i+1}) && \implies \\
&\forall 1 \leq i < \deg(v)\ \forall \pi' \in \text{pre}(\pi).\ \sigma(v)_{i+1} \in \text{Failed}(\pi') \implies \sigma(v)_i \in \text{Failed}(\pi') && \implies \\
&\forall 1 \leq i < \deg(v)\ \forall \pi' \in \text{pre}(\pi).\ \sigma(v)_{i+1} \in \text{JustFailed}(\pi') \implies \sigma(v)_i \in \text{Failed}(\pi') && \implies \\
&\text{(By induction over the length of } \pi' \text{ we get:)} \\
&\forall 1 \leq i < \deg(v)\ \forall \pi' \in \text{pre}(\pi).\ \text{Failable}(\pi') && \implies \\
&\forall 1 \leq i < \deg(v).\ \text{Failable}(\pi_{|-1}) && \square
\end{aligned}$$

We give a characterisation of a pand-gate that has just failed.

Proposition 4.12. *Let F be a DFT and $v \in F_{PAND}$. Let $\pi \in F_{BE}^{\triangleright}$ and $x \in BE$.*

$$v \in \mathit{JustFailed}(\pi \cdot x) \implies \sigma(v)_{\downarrow} \in \mathit{JustFailed}(\pi \cdot x)$$

Proof.

$$
\begin{aligned}
& v \in \text{JustFailed}(\pi \cdot x) && \Longrightarrow \\
& v \in \text{Failed}(\pi \cdot x) \wedge v \notin \text{Failed}(\pi) && \Longrightarrow \\
& \sigma(v) \subseteq \text{Failed}(\pi \cdot x) \wedge v \in \text{Failable}(\pi) \wedge v \notin \text{Failed}(\pi) && \Longrightarrow \\
& \sigma(v)_{\downarrow} \in \text{Failed}(\pi \cdot x) \wedge v \in \text{Failable}(\pi) \wedge v \notin \text{Failed}(\pi)
\end{aligned}
$$

It remains to show that

$$v \in \text{Failable}(\pi) \wedge v \notin \text{Failed}(\pi) \implies \sigma(v)_{\downarrow} \notin \text{Failed}(\pi)$$

$$
\begin{aligned}
& v \in \text{Failable}(\pi) \wedge v \notin \text{Failed}(\pi) && \Longrightarrow \\
& \text{By the monotonicity of Failable} \\
& v \in \text{Failable}(\pi) \wedge \sigma(v) \not\subseteq \text{Failed}(\pi) && \Longrightarrow \\
& \forall 1 \leq i < \deg(v).\ \text{FB}_{\pi}(\sigma(v)_i, \sigma(v)_{i+1}) \wedge \sigma(v) \not\subseteq \text{Failed}(\pi) && \Longrightarrow \\
& \forall 1 \leq i < \deg(v).\ \text{FB}_{\pi}(\sigma(v)_i, \sigma(v)_{\downarrow}) \wedge \sigma(v) \not\subseteq \text{Failed}(\pi)
\end{aligned}
$$

Now, if we assume $\sigma(v)_{\downarrow} \in \text{Failed}(\pi)$,

$$\forall 1 \leq i < \deg(v).\ \sigma(v)_i \in \text{Failed}(\pi) \wedge \sigma(v) \not\subseteq \text{Failed}(\pi)$$

which is a contradiction. Thus, $\sigma(v)_{\downarrow} \notin \text{Failed}(\pi)$. □

Corollary 4.13. *Let F be a DFT and $v \in F_{PAND}$. Let $\pi \in F_{BE}^{\triangleright}$.*

$$v \in \mathit{JustFailed}(\pi) \implies \sigma(v)_{\downarrow} \in \mathit{JustFailed}(\pi)$$

Proof. Proposition 4.12 covers $\pi \neq \varepsilon$. We just have to show

$$v \in \text{JustFailed}(\varepsilon) \implies \sigma(v)_{\downarrow} \in \text{JustFailed}(\varepsilon),$$

which follows directly as the successors cannot fail before ε. □

Before we continue with spare gates, we formally show that only elements in the predecessor-closure of a given-failure element can be in the set of failed elements after $\pi = \varepsilon$.

Proposition 4.14. *It holds that*

$$\mathit{Failed}(\varepsilon) \subseteq \bigcup_{c \in F_{\text{CONST}(\top)}} \theta^*(c)$$

Proof. Assume $X = \{v \in V \mid v \in \text{Failed}(\varepsilon) \wedge v \notin \bigcup_{c \in F_{\text{CONST}(\top)}} \theta^*(c)\}$. It follows that $\forall v \in X.\ F_{\text{CONST}(\top)} \cap \sigma^*(v) = \emptyset$. By Lemma 4.1 we know that for $v \in F_{\text{CONST}(\bot)} \cup F_{\text{BE}} \cup F_{\text{SPARE}}$, $v \notin \text{Failed}(\varepsilon)$, and thus $v \notin X$. Take now a $v \in X$. By Proposition 4.10,for $v \in F_{\text{AND}} \cup F_{\text{OR}} \cup F_{\text{VOT}(k)}$, and by Corollary 4.13 for $v \in F_{\text{PAND}}$, it holds that $v' \in \sigma(v)$, $v' \in X$. However, as no leaf nodes can be in X, either X is infinite or empty. As DFTs are finite, X is empty. □

We are now going to do characterise failure-propagation for spare-gates. As with pand-gates, we start with an alternative characterisation for spare-gates and then characterise a spare-gate which has just failed. As preparation, we start with a observation based on the fact that a spare-gate only claims one spare module at a time.

Lemma 4.15. *Let F be a DFT with $s \in F_{SPARE}$ and $\pi \in F_{BE}^{\triangleright}$. Let $m \in \mathbb{N}$ s.t. $\sigma(s)_m = \mathit{NextSpare}(\pi, s)$. It holds that*

$$\arg\max_i \{\sigma(s)_i \mid \sigma(s)_i \in \mathit{Claimed}(\pi_{|-1}, s)\} < j$$

Proof. Assume that there exists $k > m$, $\sigma(s)_k \in \text{Claimed}(\pi_{|-1})$, then there exists $\pi' \in \text{pre}(\pi)$ such that $\sigma(s)_k = \text{NextSpare}(\pi', s)$. We have that $\text{NextSpare}(\pi', s) = \min_j\{\sigma(v)_j \in V \mid \sigma(v)_j \notin \text{Failed}(\pi') \wedge \text{ClaimedBy}(\pi''_{|-1}, \sigma(v)_j) = \bot\}$. As $m < k$, it holds that either $\sigma(s)_m \in \text{Failed}(\pi')$ or $\text{ClaimedBy}(\pi'_{|-1}, \sigma(v)_m) \neq \bot$. This is a contradiction to coherency, as $\sigma(v)_m = \text{NextSpare}(\pi, s)$, which yields $\sigma(s)_m \notin \text{Failed}(\pi)$ and $\text{ClaimedBy}(\pi_{|-1}, \sigma(v)_m) = \bot$. □

We define the unique element which was last claimed by a spare-gate.

$$\text{LastClaimed}\colon F_{\text{BE}}^{\triangleright} \to (F_{\text{SPARE}} \to V)$$
$$\text{LastClaimed}(\pi)(s) = \max_i\{\sigma(s)_i \in \text{Claimed}(\pi, s)\}$$

Proposition 4.16. *Let F be a DFT with $s \in F_{SPARE}$ and $\pi \in F_{BE}^{\triangleright}$, then*

$$\forall v \in \mathit{Claimed}(\pi, s)\ \exists \pi' \in \mathit{pre}(\pi) \cup \{\pi\}.\ v = \mathit{LastClaimed}(\pi, s).$$

Proof. By definition, $\text{Claimed}(\pi, s) \subseteq \sigma(s)$. For $v = \sigma(s)_1$, by Corollary 4.3, $\text{Claimed}(\varepsilon, s) = \{v\}$, so $\text{LastClaimed}(\varepsilon, s) = v$. For $v \in \text{Claimed}(\pi, s) \setminus \{\sigma(s)_1\}$, there exists a $\pi' \in \text{pre}(\pi)$ s.t. $v \notin \text{Claimed}(\pi'_{|-1}, s)$ and $v \in \text{Claimed}(\pi', s)$. It follows that $v = \text{NextSpare}(\pi', s)$. With Lemma 4.15, it follows that $\max_i\{\sigma(s)_i \in \text{Claimed}(\pi', s)\} = v$. □

Furthermore, we observe that primary modules do not fail at initialisation.

Corollary 4.17. *Let F be a DFT and $s \in F_{SPARE}$. It holds that*

$$\sigma(s)_1 \notin \mathit{Failed}(\varepsilon)$$

Proof. We consider $X = \mathsf{SM}_{\sigma(s)_1}$. By a structural induction over the graph of F, restricted to the elements in X, we show that $\text{Failed}(\varepsilon) \cap X = \emptyset$. The types of the sinks in the DAG are either the usual leaf-types or spare-gates. Notice that given-failure elements are not in the module, as by Definition 4.9 we have that $F_{\text{CONST}(\top)} \cap \mathsf{SM}_{\sigma(s)_1} = \emptyset$. Following Lemma 4.1, none of the possible sinks is initially failed. The possible gate-types are the usual gate-types, but without the spare-gates. Again by Lemma 4.1, none of these fail if no successor has failed. By structural induction, we have that no element in X fails initially, and thus $\sigma(s)_1$ does not fail at initialisation. □

Lemma 4.18. *Let F be a DFT and $s \in F_{SPARE}$, furthermore let $\pi \in F_{BE}^{\triangleright} \setminus \{\varepsilon\}$,*

$$s \in \mathit{Failed}(\pi) \iff \mathit{LastClaimed}(\pi_{|-1}, s) \in \mathit{Failed}(\pi) \wedge \mathit{Available}(\pi)(s) = \emptyset.$$

Proof. We only have to show

$$\text{Claimed}(\pi_{|-1}, s) \subseteq \text{Failed}(\pi) \iff \text{LastClaimed}(\pi_{|-1}, s) \in \text{Failed}(\pi)$$

Notice that from left to right is trivial as it is a specialisation. Furthermore, for $\text{LastClaimed}(\pi, s) = \sigma(s)_1$, we have that $\text{Claimed}(\pi, s) = \{\sigma(s)_1\}$, and we are done. Now, we assume that

$$\text{Claimed}(\pi_{|-1}, s) \not\subseteq \text{Failed}(\pi),$$

which means

$$\begin{aligned}
&\text{Claimed}(\pi_{|-1}, s) \not\subseteq \text{Failed}(\pi) && \Longrightarrow\\
&\exists v \in \text{Claimed}(\pi_{|-1}, s).\ v \notin \text{Failed}(\pi) && \Longrightarrow\\
&\exists v \in \text{Claimed}(\pi_{|-1}, s) \setminus \{\text{LastClaimed}(\pi_{|-1}, s)\}.\ v \notin \text{Failed}(\pi)
\end{aligned}$$

As $v \neq \text{LastClaimed}(\pi_{|-1}, s)$, there must be a $\pi' \in \text{pre}(\pi_{|-1})$ with $\text{LastClaimed}(\pi'_{|-1}, s) = v$ (cf. Proposition 4.16). Moreover, we can choose π' as above s.t. s is claiming, as otherwise, $v = \text{LastClaimed}(\pi_{|-1}, s)$.

For s claiming at π' we have $\text{Claimed}(\pi'_{|-1}) \subseteq \text{Failed}(\pi')$ (cf. Definition 4.12), thus especially $v \in \text{Failed}(\pi')$, and thus, by coherency, $v \in \text{Failed}(\pi)$, which is a contradiction. Thus,

$$\text{Claimed}(\pi_{-1}, s) \subseteq \text{Failed}(\pi).$$

□

Proposition 4.19. *Let F be a DFT and $v \in F_{SPARE}$. Let $\pi \in F_{BE}^{\triangleright}$ and $x, y \in BE$.*

$$v \in \mathit{JustFailed}(\pi \cdot xy) \implies \mathit{LastClaimed}(\pi \cdot x) \in \mathit{JustFailed}(\pi \cdot xy)$$

Proof.

$$\begin{aligned}
&v \in \text{JustFailed}(\pi \cdot xy) && \implies \\
&v \in \text{Failed}(\pi \cdot xy) \wedge v \notin \text{Failed}(\pi \cdot xy) && \implies \\
&\text{LastClaimed}(\pi \cdot x) \in \text{Failed}(\pi \cdot xy) \wedge v \notin \text{Failed}(\pi \cdot xy)
\end{aligned}$$

As we have

$$\begin{aligned}
&\text{LastClaimed}(\pi \cdot x, v) \in \text{JustFailed}(\pi \cdot xy) \iff \\
&\quad \text{LastClaimed}(\pi \cdot x, v) \in \text{Failed}(\pi \cdot xy) \wedge \text{LastClaimed}(\pi \cdot x, v) \notin \text{Failed}(\pi \cdot x)
\end{aligned}$$

it suffices to show

$$v \notin \text{Failed}(\pi \cdot xy) \implies \text{LastClaimed}(\pi \cdot x, v) \notin \text{Failed}(\pi \cdot x).$$

Therefore,

$$\begin{aligned}
&v \notin \text{Failed}(\pi \cdot xy) && \implies \\
&\text{LastClaimed}(\pi, v) \notin \text{Failed}(\pi \cdot x) \vee \text{Available}(\pi \cdot x) \neq \emptyset && \implies \\
&\text{LastClaimed}(\pi, v) \notin \text{Failed}(\pi \cdot x) \vee \\
&\quad (\text{LastClaimed}(\pi, v) \in \text{Failed}(\pi) \wedge \text{Available}(\pi \cdot x) \neq \emptyset) && \implies \\
&(\text{LastClaimed}(\pi, v) = \text{LastClaimed}(\pi \cdot x, v) \wedge \text{LastClaimed}(\pi, v) \notin \text{Failed}(\pi \cdot x)) \vee \\
&(\text{LastClaimed}(\pi, v) \in \text{Failed}(\pi) \wedge \text{Available}(\pi \cdot x) \neq \emptyset) && \implies \\
&\text{LastClaimed}(\pi \cdot x, v) \notin \text{Failed}(\pi \cdot x) \vee \\
&\quad (\text{LastClaimed}(\pi, v) \in \text{Failed}(\pi) \wedge \text{Available}(\pi \cdot x) \neq \emptyset) && \implies \\
&\text{LastClaimed}(\pi \cdot x, v) \notin \text{Failed}(\pi \cdot x) \vee \text{LastClaimed}(\pi \cdot x, v) = \text{NextSpare}(\pi \cdot x, v) && \implies \\
&\text{LastClaimed}(\pi \cdot x, v) \notin \text{Failed}(\pi \cdot x)
\end{aligned}$$

□

Corollary 4.20. *Let F be a DFT and $v \in F_{SPARE}$. Let $\pi \in F_{BE}^{\triangleright}$ and $x, y \in BE$.*

$$v \in \mathit{JustFailed}(\pi \cdot x) \implies \mathit{LastClaimed}(\pi, v) \in \mathit{JustFailed}(\pi \cdot x)$$

Proof. Proposition 4.19 covers $\pi \neq \varepsilon$. We just have to show

$$v \in \text{JustFailed}(x) \implies \text{LastClaimed}(\varepsilon, v) \in \text{JustFailed}(x).$$

We have that $\text{LastClaimed}(\varepsilon, v) = \sigma(v)_1$. It is clear that $v \in \text{JustFailed}(x) \implies \sigma(v)_1 \in \text{Failed}(x)$. By Corollary 4.17, we have that $\sigma(v)_1 \notin \text{Failed}(\varepsilon)$, which completes the proof. □

We can observe that failure propagation always is caused by an immediate successor in the DFT.

Proposition 4.21. *Let F be a DFT with an element $v \notin F_{BE} \cup F_{FDEP}$, and $\pi \in F_{BE}^{\triangleright}$. The following holds:*

$$v \in \mathit{JustFailed}(\pi) \implies \exists v' \in \sigma(v).\ v \in \mathit{JustFailed}(\pi).$$

We omit the proof here as the cases $v \in F_{\text{PAND}}$ and $v \in F_{\text{SPARE}}$ are implied by respectively Corollary 4.13, and Corollary 4.20. and the other cases are straight-forward applications of the definitions of Failed and JustFailed.

Corollary 4.22. *Let F be a DFT with an element $v \notin F_{BE} \cup F_{FDEP}$, and $\pi \in F_{BE}^{\triangleright}$. The following holds:*

$$v \in \mathit{Failed}(\pi) \implies \exists v' \in \sigma(v).\ v \in \mathit{Failed}(\pi).$$

Proof. Follows immediately from Proposition 4.21, coherency and that $\text{JustFailed}(\pi) \subseteq \text{Failed}(\pi)$. □

The following definition and corollaries help us to streamline further arguments based on the reasoning above.

Definition 4.14 (Immediate failure-cause path)**.** Given a DFT F with an element v, $v \notin F_{\text{FDEP}}$ and a basic event e, each directed path $p = x_1 \dots x_m$ from $v = x_1$ to $e = x_m$ is called an *immediate failure-cause path for v from e* if

$$\forall i.\ x_i \in F_{\text{PAND}} \implies x_{i+1} = \sigma(x_i)_{\downarrow}$$ ■

The set $\text{ifcp}(v) = \{p \mid \exists e \in F_{\text{BE}} \text{s.t. } p \text{ is a immediate failure-cause path for } v \text{ from } e\}$ is the set of all immediate failure cause paths for v.

Remark 24. Delayed failure-cause paths are introduced in Section 4.4 on page 102.

Corollary 4.23. *Let F be a DFT with element v, $v \notin F_{FDEP}$.*

$$\forall p \in \mathit{ifcp}(v).\ p \subseteq \sigma^*(v)$$

Proof. Each directed path only visits successors. □

Corollary 4.24. *Let F be a DFT with an element v, and $\pi \in F_{BE}^{\triangleright} \setminus \{\varepsilon\}$. The following holds:*

$$v \in \mathit{JustFailed}(\pi) \implies \exists p \in \mathit{ifcp}(v). \forall v' \in p.\ v' \in \mathit{JustFailed}(\pi) \wedge p_{\downarrow} = \pi_{\downarrow}$$

The proof is a successive application of Proposition 4.21(each element which has just failed has a child which has just failed) and Corollary 4.13(for pand-gates the last child has just failed), together with the fact that $\pi_{\downarrow}$ is the only basic event which has just failed.

Corollary 4.25. *Let F be a DFT with an element v. The following holds:*

$$v \in \mathit{Failed}(\varepsilon) \implies \exists p \in \mathit{ifcp}(v). \forall v' \in p.\ v' \in \mathit{Failed}(\pi) \wedge p_{\downarrow} \in \mathit{CONST}(\top).$$

Definition 4.15. Given a DFT F with elements V and $\pi \in F_{\text{BE}}^{\triangleright} \setminus \{\varepsilon\}$. Let $v \in V$ and $v_1 \dots v_m = p \in \text{ifcp}(v)$. If for all $1 \leq i < m$, $v_i \in F_{\text{SPARE}} \implies v_{i+1} = \text{LastClaimed}(\pi_{|-1}, v_i)$, we call p an *immediate failure cause path at π*. The set $\text{ifcp}_{\pi}(v) \subseteq \text{ifcp}_{\pi}(v)$ is the set of all immediate failure-cause paths at π. ■

The following follows then directly from Corollary 4.24 and Corollary 4.20.

Corollary 4.26. *Let F be a DFT with an element v, and $\pi \in F_{BE}^{\triangleright} \setminus \{\varepsilon\}$. The following holds:*

$$v \in \mathit{JustFailed}(\pi) \implies \exists p \in \mathit{ifcp}_{\pi}(v). \forall v' \in p.\ v' \in \mathit{JustFailed}(\pi) \wedge p_{\downarrow} = \pi_{\downarrow}$$

The following corollaries are a direct consequence from the restrictions in Definition 4.9.

Corollary 4.27. *Let F be a DFT and $r \in \mathsf{EMR}_F$.*

$$\forall v \in \mathsf{SM}_r.\ \theta^*(v) \subseteq \sigma^*(r) \cup \mathsf{EM}_r$$

Corollary 4.28. *Let $F = (V, \sigma, \mathit{Tp}, \Theta, \mathit{top})$ be a DFT and $r \in \mathsf{SMR}_F$ and $v \in V \setminus \mathsf{SM}_r$. Then the following statement holds*

$$\forall p \in \mathit{ifcp}(v)\ \forall x \in V\ x \in p \implies r \in p \wedge \exists s \in \theta(r) \cap F_{\mathit{SPARE}}.\ s \in p$$

Corollary 4.29. *Given a DFT F with element v, it holds that*

$$\exists s \in F_{SPARE}.\ v \in \sigma^*(s) \implies \exists r \in \mathsf{SMR}_F.\ v \in \mathsf{SM}_r$$

Theorem 4.30. *Let F be a DFT and $\pi \in F^{\triangleright}_{BE}$. At most one spare gate is claiming after π.*

The proof will also show this immediate consequence.

Corollary 4.31. *Let F be a DFT with elements V and $\pi \in F^{\triangleright}_{BE}$, it holds that*

$$\forall v \in V.\ |\mathit{ClaimedBy}(\pi, v)| \leq 1$$

Proof of Theorem 4.30. We use an induction over the length of π, with which we simultaneously show the statement of the theorem and of Corollary 4.31.

For $\pi = \varepsilon$, by definition of claiming, no spare-gate is claiming. Furthermore, we have that all spare-gates have claimed their primary component. By the restrictions in Definition 4.9, it follows that $|\text{ClaimedBy}(\pi, v)| \leq 1$. By Corollary 4.17, we have that these elements have not failed.

Let $\pi = \pi' \cdot x$. Consider x. Either $\forall s \in F_{\text{SPARE}}.\ x \notin \sigma^*(s)$. Then $\forall s \in F_{\text{SPARE}}, s \notin \text{JustFailed}(\pi \cdot x)$ and $s \notin \text{Claiming}(\pi \cdot x)$. Thus we only have to consider $\exists s \in F_{\text{SPARE}}.\ x \in \sigma^*(s)$. By Corollary 4.29, we have $\exists r \in \mathsf{SMR}_F.\ x \in \mathsf{SM}_r$. We consider such a SM_r.

- Either $r \notin \text{JustFailed}(\pi' \cdot x)$, but then, we claim, $\text{JustFailed}(\pi') \subset \mathsf{SM}_r$. To show this, assume $\exists v \in \text{JustFailed}(\pi) \setminus \mathsf{SM}_r$. By Corollary 4.26, there must be a directed path p from v to x, s.t. $\forall y \in p.\ y \in \text{JustFailed}(\pi \cdot x)$. Following Corollary 4.28, we have that for all such p, $r \in p$, but as $r \notin \text{JustFailed}(\pi \cdot x)$, we have a contradiction. Thus $\text{JustFailed}(\pi) \subset \mathsf{SM}_r$. Using Corollary 4.27 and Corollary 4.26, we have that $\text{JustFailed}(\pi) \cap \mathsf{SMR}_F = \emptyset$, and thus, no spare-gate is claiming.
- Now $r \in \text{JustFailed}(\pi' \cdot x)$. By the induction hypothesis, we know that $|\text{ClaimedBy}(\pi, r)| \leq 1$. If $\text{ClaimedBy}(\pi', r) = \emptyset$, then there exists no spare gate which is claiming or has just failed. The remaining case is thus $\text{ClaimedBy}(\pi', r) = \{s\}$ for some $s \in F_{\text{SPARE}}$, which we cover in the steps below.
  1. We observe that for
  $$R = \{r' \in V \mid r' \in \sigma(s) \wedge r' \neq r\} \subseteq \mathsf{SMR}_F,$$
  it holds that
  $$R \cap \text{JustFailed}(\pi' \cdot x) = \emptyset.$$
  Again, we assume to the contrary that there exists an $r' \in R$ s.t. $r' \in \text{JustFailed}(\pi' \cdot x)$. Then there exists a p directed path from r' to x. As before, for all such p, we have that $r \in p$. If $r \in p$, then certainly an $s' \in \theta(r)$ such that $s' \in p$ and $s' \in F_{\text{SPARE}}$. If $s' = s$, then the DFT has a cycle, so $s' \neq s$. Now for such a $s' \in \text{JustFailed}(\pi' \cdot x)$, $\text{LastClaimed}(\pi', s') \in \text{JustFailed}(\pi')$. If $\text{LastClaimed}(\pi', s') = r$, then $\text{ClaimedBy}(\pi', r) \subseteq \{s, s'\}$, which is impossible by the induction hypothesis. Thus, $\text{LastClaimed}(\pi'_{|-1}, s') \neq r$. By repeating this argument on the path from $\text{LastClaimed}(\pi'_{|-1}, s')$ we either have infinitely many spare-gates which have just failed, or a cycle.
  2. If $r \neq \text{LastClaimed}(\pi', s)$, then s neither fails or claims, as no other successor of s fails, see (1).
  3. If $s \notin \text{JustFailed}(\pi' \cdot x)$, then $\text{JustFailed}(\pi' \cdot x) \subseteq \mathsf{SM}_r$. Let $K = \theta(r) \cap F_{\text{SPARE}} \cap \text{JustFailed}(\pi' \cdot x)$. We only have to show $K = \emptyset$, as all directed paths which go from any $v \notin \mathsf{SM}_r$ to x have to go through r and through one of the spare gates in $\theta(r)$, cf. (1).
  Now assume that $\exists s' \in K, s' \neq s$. As $r \neq \text{LastClaimed}(\pi', s')$, there must be a directed path from $\text{LastClaimed}(\pi', s')$ to x via some $\hat{s}_1 \in K$. We know that $s' \neq \hat{s}_1$, as the DFT is acyclic. Now for $\hat{s}_1$, by repeating the argument, we get that there must be a $\hat{s}_2 \in X$, with $\hat{s}_2 \notin \{s', \hat{s}_1\}$. Thus K is either infinite or empty, and as the DFT is finite, X is empty.
  4. Following (2), we assume $r = \text{LastClaimed}(\pi', s)$. Furthermore s cannot fail and claim at the same time (cf. Proposition 4.8). Assume that s claims, then we have one spare-

gate claiming. However, following (3), no other spare gates fail. Now s claims a single node v, which – by definition – has not been claimed before, so $\text{ClaimedBy}(\pi' \cdot x) = \{v\}$.

5. Assume s fails. We have that

$$(\theta(r) \setminus \{s\}) \cap \text{JustFailed}(\pi' \cdot x) \subseteq \theta^*(s)$$

by a similar argument as in (4). Thus, for all $v \in \text{JustFailed}(\pi')$, either $v \in \mathsf{SM}_r$, or

$$\forall p \in \text{ifcp}'_\pi(v).\ \forall v' \in p.\ v' \in \text{JustFailed}(\pi' \cdot x) \implies s \in p$$

Now either $\theta^*(s) \cap F_{\text{SPARE}} = \emptyset$, and we are done, or $s \in \mathsf{SM}_{r'}$ for some $r' \in \mathsf{SMR}$. But then, we can consider $\mathsf{SM}_{r'}$ like SM_r with an analogous argument, ignoring elements in SM_r and using s instead of x as target for all failure paths. Again, the next spare either claims or fails, but if it claims, no further spare gates fail or claim.

□

The following propositions are helpful handling spare gates. The follow from the proof above.

Proposition 4.32. *Given a DFT F, and a event trace π such that a $v \in F_{SPARE}$ is the (unique) spare-gate that is claiming. Let S be the set of spares that have just failed after π, i.e. $S = \{s \in F_{SPARE} \mid s \in \mathit{JustFailed}(\pi)\}$. It holds that $S \subseteq \sigma^*(v)$.*

Proposition 4.33. *Given a DFT F with a spare-gate $s \in F_{SPARE}$. Let $z = \mathit{LastClaimed}(\pi_{|-1})(s)$.*

$$z \in \mathit{JustFailed}(\pi) \implies \forall z' \in \sigma(s) \setminus \{z\}.\ z' \notin \mathit{JustFailed}(\pi)$$

Defining the internal state of a DFT The *internal state* of a DFT F contains all information to deduce the effect of any following failure of a basic event.

Definition 4.16. Given a DFT F and $\pi = F^{\triangleright}_{\text{BE}}$, the *(internal) state* after the occurrence of π is a tuple $\text{State}_{\text{int}}(\pi) = (\text{Failed}(\pi), \text{Failable}(\pi), \text{ClaimedBy}(\pi))$. ■

The internal state $\text{State}_{\text{int}}(\pi \cdot x)$ depends on F, x and $\text{State}_{\text{int}}(\pi)$, but not on π itself. We formalise this in the following statement.

Proposition 4.34. *Given a DFT F and $\pi \cdot x, \pi' \cdot x \in F^{\triangleright}_{BE}$. If $\mathit{State}_{int}(\pi) = \mathit{State}_{int}(\pi')$, then $\mathit{State}_{int}(\pi \cdot x) = \mathit{State}_{int}(\pi' \cdot x)$.*

We omit a proof here. The proof involves rewriting every item in $\text{State}_{\text{int}}(\pi \cdot x)$ in terms of F, x and $\text{State}_{\text{int}}(\pi)$.

4.2.5. Towards functional-complete event chains

In the semantics introduced above, we simply ignored functional dependencies and only discussed basic events. Here, we extend the semantics to component failures and functional dependencies.

Intuition Each component failure causes a basic event to fire. Moreover, functional dependencies may also trigger dependent basic events, that is, after the trigger of the functional dependency has failed, dependent basic events fail.

While gates fail immediately with their triggering event, we define functional dependencies such that they fail ordered. As we assume that only infinitesimal time passes to trigger functional dependencies, we require all functional dependencies to fire before any non-dependent event fails. This leads to an alternating sequence of basic events attached to component failures and possibly empty sequences of basic events which are triggered by functional dependencies. The possible paths can be grouped to form a qualitative model. This model covers the DFT model up to the disabled basic events as their attached component failures are in cold standby, which we cover in the next section.

Formal specification We start with a formalisation of the dependent basic events, and then discuss an automaton which provides, given a failure trace, all possible and corresponding event traces. Functional dependencies are gates in the DFT syntax, but it is more convenient to have the direct relation.

$$\text{DepEvents}(v) = \bigcup_{v' \in \{w \in \text{FDEP} \mid \sigma(w)_1 = v\}} \{\sigma(v')_2\}$$

For $v' \in \text{DepEvents}(v)$, we write $v \rightsquigarrow v'$, we refer to v as the *trigger* and to v' as the *dependent event*. Please, notice that we can have cyclic dependencies, either direct or indirect, e.g. we could have $v_1 \rightsquigarrow v_2$ and $v_2 \rightsquigarrow v_1$.

Definition 4.17. Given a event chain π, the set of *dependent* or *triggered events* is

$$\Delta(\pi) = \Big(\bigcup_{v \in \text{Failed}(\pi)} \text{DepEvents}(v) \Big) \setminus \{v \in \pi\}$$ ■

We introduce the *functional (dependency) transducer*, which returns for a given failure trace the possible event traces.

Definition 4.18 (Functional Dependency transducer)**.** Given a DFT $F^\Omega = (V, E, \text{Tp}, \Theta, \text{top})$, we define the transducer $\mathcal{T}_F = (Q, \Sigma, \Gamma, I, \circledcirc, \delta)$

- State space $Q = \Omega^\triangleright \times F_{\text{BE}}^\triangleright$,
- Input alphabet $\Sigma = \Omega$.
- Output alphabet $\Gamma = F_{\text{BE}}$.
- Initial states $I = \{(\varepsilon, \varepsilon)\}$.
- Final states $\circledcirc = \{(\rho, \pi) \mid \Delta(\pi) = \emptyset\}$.
- Transition relation $\delta \subseteq Q \times \Sigma \times \Gamma \times Q$, with

$$\begin{aligned}
\delta = &\{((\rho,\pi), \omega, \Theta(\omega), (\rho \cdot \omega, \pi \cdot \Theta(\omega))) && \mid \omega \notin \rho \wedge \Delta(\pi) = \emptyset \wedge \Theta(\omega) \notin \pi\} && (4.1)\\
\cup &\{((\rho,\pi), \varepsilon, x, (\rho, \pi \cdot x)) && \mid x \in \Delta(\pi)\} && (4.2)\\
\cup &\{((\rho,\pi), \omega, \varepsilon, (\rho \cdot \omega, \pi)) && \mid \omega \notin \rho \wedge \Delta(\pi) = \emptyset \wedge \Theta(\omega) \in \pi\} && (4.3)
\end{aligned}$$

$\mathcal{T}$ is called the *functional transducer of* F. ■

The state space is constructed as any combination of failure and event trace. The input alphabet consists of the possible component failures, while the output alphabet consists of the possible (basic) events. The initial state describes the state in which no component has failed and no event has failed. We mark any state as accepting in which the set of dependent events is empty. We call these final states *resolved*, and all others *unresolved*. The transition relation consists of three types of transitions. The transitions connect the states such that the reachable state space is a tree. The state identifiers correspond to the concatenation of all transitions. All these transitions share that they do not allow component failures to occur twice, hence $\omega \notin \rho$. The transition set consists of three subsets, which we discuss below:

1. The first set encodes those cases in which there are no functional dependent events, and a component failure occurs whose attached basic event has not been internally triggered by a functional dependency. It causes the attached basic event to fail.
2. The second set describes the cases in which there are functional dependent events. In that case, without a component failure, an event may be triggered internally. Here, $\omega \notin \rho$ is implicitly encoded, cf. Definition 4.17.
3. The third set consists of those cases in which the attached events of a component failure have been internally triggered, by some transition from (2). The component failure does not affect on the internal state of the DFT.

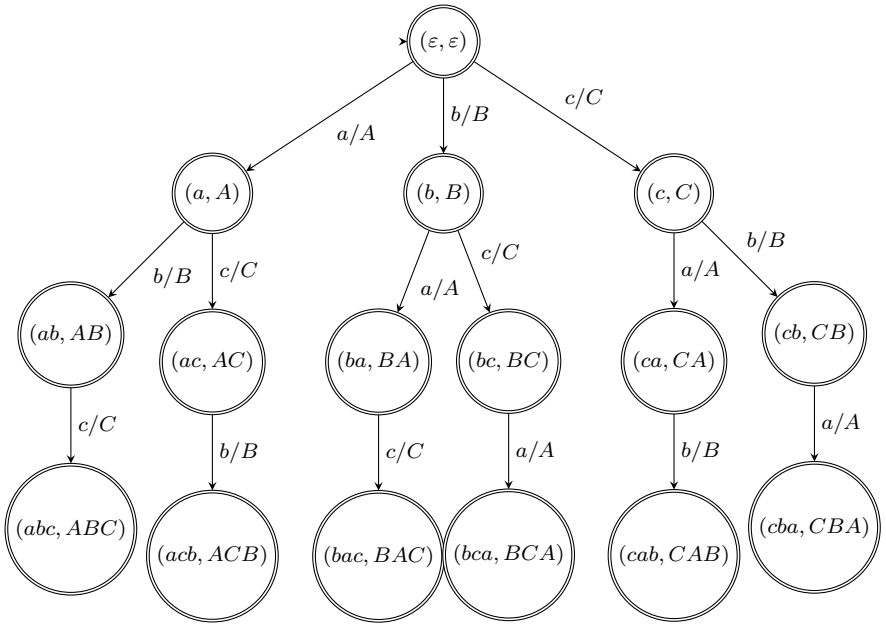


Figure 4.3.: The reachable fragment of the functional transducer from Example 4.3.

Please notice that the reachable fragment of $\mathcal{T}_F$ is much smaller than the total state space. We write $\mathcal{T}(\rho)$ to denote $\{\pi \mid (\rho, \pi) \in \mathcal{L}(\mathcal{T})\}$

Examples

Example 4.3. The functional transducer of the fault tree from Example 4.1 on page 75 is depicted in Figure 4.3. We notice that the DFT doesn't contain any functional dependencies. Therefore, the set of dependent events is empty after any event trace. All events are governed by the occurrence of component failures, which are not restricted. The constructed transducer therefore describes all possible sequences of these component failures. ▲

Example 4.4. The functional transducer of the fault tree from Example 4.2 on page 76 is depicted in Figure 4.4. Several functional dependencies are present in the DFT. Initially, none of elements has failed, and thus, none of the events is triggered. The first occurrence of a basic event is thus governed by the first occurrence of a component failure. We consider all three transitions separately.

- Let us first consider the case that c occurs first. With c, basic event C occurs, but no further basic events are triggered. Thus, next, the transducer waits for another component failure. This can be either a or b, but not c.
- Now, if we consider the occurrence of b (with B failing), then a functional dependency with dependent event C is triggered. As there exists a dependent event for the event trace B, the component failures are not (yet) enabled, instead, we first have to execute C (without a component failing, hence ε/C).
- The last remaining option is the occurrence of a (with A). Two functional dependencies are triggered. The transducer can either first execute B or C. W.l.o.g., we assume here that it takes B. Now, the set of dependent events for the trace AB is still not empty, therefore, the transition ε/C exists.

Furthermore, we consider the state a/ABC as obtained in the path described above. Here, component failures of b and c can still occur, but do not have an effect on the DFT, as the attached basic events have already failed. ▲

4.2.6. Activation

As discussed before, we need to enhance our model with information about the active and inactive components. We do not embed this information into the qualitative model, but instead add this

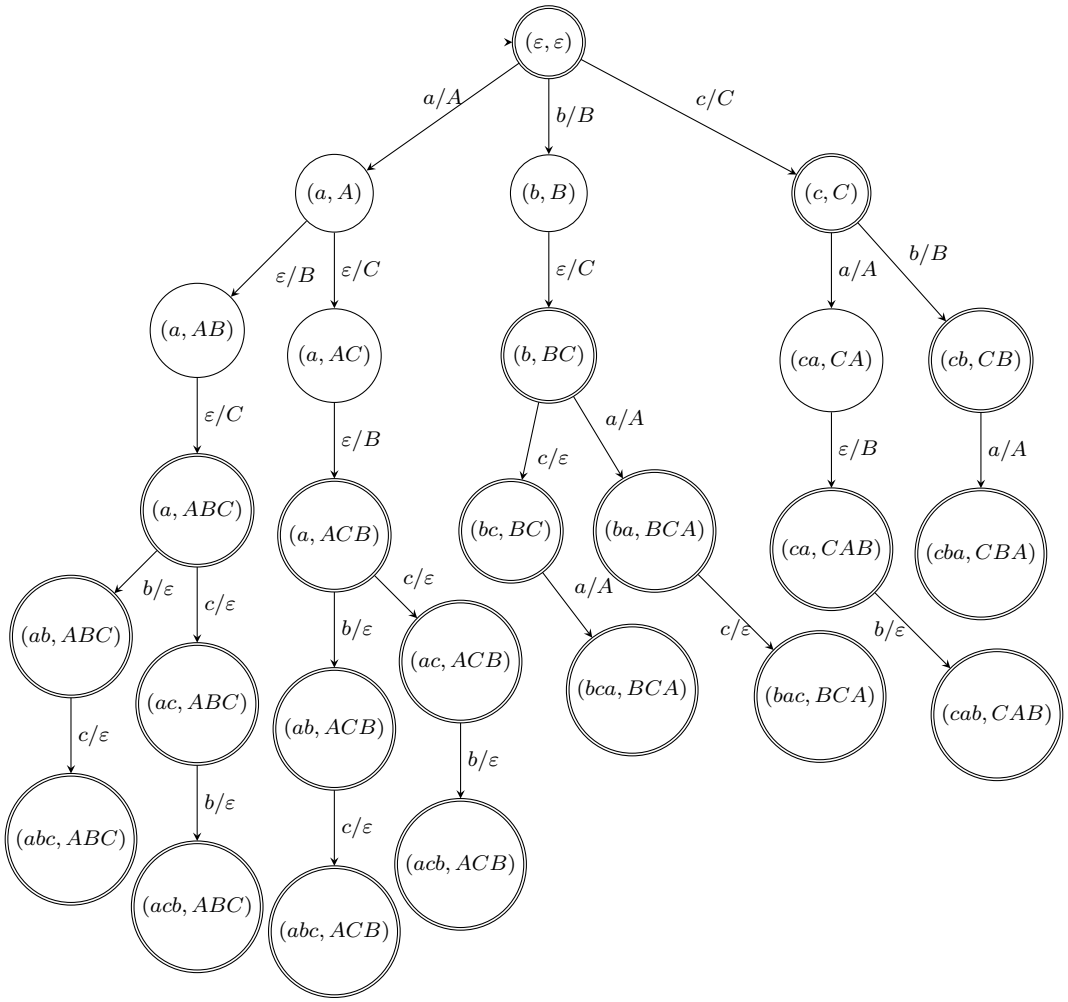


Figure 4.4.: The reachable fragment of the functional transducer from Example 4.4.

information only during the transition from the qualitative model to the quantitative model.

Intuition Components are activated if their attached basic event is part of an active module. The module represented by top is always active. Other modules are active from the moment

- their representative is claimed by an active spare, or
- the spare which claimed their representative becomes active,

whatever happens first. This corresponds to early claiming with late activation, as discussed in Section 3.3.4.4 on page 44.

Formal specification We are only interested in the activation of component failures, therefore, we introduce the mapping $\text{Activated}\colon F_{\text{BE}}^{\triangleright} \to \mathcal{P}(\Omega)$.

In order to ease the definition of this mapping, we introduce the auxiliary Active as the smallest fixpoint such that:

$$\text{Active}\colon F_{\text{BE}}^{\triangleright} \to \mathcal{P}(V)$$
$$\text{Active}(\pi) = \mathsf{EM}_{\text{top}} \cup \bigcup \{\mathsf{SM}_r \mid \text{ClaimedBy}(\pi, r) \in \text{Active}(\pi)\}$$

With this notion, a component (failure) is *activated* whenever its corresponding basic event is active.

$$\text{Activated}(\pi) = \{\omega \in \Omega \mid \Theta(\omega) \in \text{Active}(\pi)\}$$

Examples We describe the activation mechanism for our two running examples.

Example 4.5. We consider Example 4.1 on page 75. The functional transducer of this is described in Example 4.3 on page 93. The two spare gates are in the module of the top-level, and therefore are active. The two primary components (a, b) are claimed by active components, and therefore are also active. The spare module (c) is not active. It will become active as soon as it is claimed by an active spare. As both spare gates are already active, we only have to consider the moment that c is claimed. c is claimable as long as it has not failed, and is not yet claimed by any spare. It is claimed by the first spare that is claiming, (radio1) or (radio2), which depends on either (a) or (b) failing. Thus considering the functional transducer, it is claimed in the states (a,a), (b,b), active in all successor states, and never activated in a state where c has occurred before. ▲

Example 4.6. We consider Example 4.2 on page 76.

As we do not have spare-gates in this example, all basic events are connected to the top level element without interference of a spare-gate. Thus, all component failures are active right from the start. ▲

We consider an additional example, to discuss a case with nested spares.

Example 4.7. We reconsider the example DFT given in Figure 3.19b on page 45, which was also used in Section 3.3.4.4 on page 44 to discuss early claiming with late activation. We see that initially, the SF spare-gate is active as it is the top. Therefore, also the primary module of SF is active, i.e., R_1, A_2 and P_1 are active. Again, as the spare-gate P_1 is active, also its primary component PA_1 is active. If we consider the failure of just PA_2, P_2 has to claim a new spare module (B_1 in this case). However, as P_2 is not active, B_1 is not activated. If we consider a subsequent failure of A_2, then SF claims a spare module. As SF is active already, P_2 is now activated, and as P_2 is now active, also B_1 is activated. ▲

4.2.7. From qualitative to quantitative

In this section, we construct the underlying quantitative model, which enables us to define probabilistic measures on the DFT. The model is constructed by transforming the transducer into a Markov automaton, and additionally taking active and inactive component failures into account.

Intuition We transform each state in the functional transducer to a state in a Markov automaton. Transitions which are due to triggered events are fired immediately and are therefore represented by immediate transitions. Transitions governed by a component failure occur as soon as the component fails, which happens with a fixed rate. This rate is either the active or the passive failure rate, which depends on the outgoing state.

Formal specification The qualitative model for a DFT F is defined via the functional transducer of F.

Definition 4.19 (Computation Tree)**.** Given a DFT F. Let $\mathcal{T}_F = (Q, \Sigma, \Gamma, I, \circledcirc, \delta)$ be the functional transducer of F. We define the labelled Markov automaton $\mathcal{C}_F = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$.

- $S = Q$.
- $\iota = I$.
- $A = \{\tau_x \mid x \in \text{BE}_F\}$
- $\hookrightarrow = \{(s, \tau_x, (s' \mapsto 1)) \mid (s, \varepsilon, x, s') \in \delta\}$
- $\dashrightarrow = \{(s, \mathcal{R}(\omega), s') \mid (s, \omega, \Theta(\omega), s') \in \delta \wedge \Theta(\omega) \in \text{Activated}(s_2)\} \cup$
  $\{(s, \alpha(\omega) \cdot \mathcal{R}(\omega), s') \mid (s, \omega, \Theta(\omega), s') \in \delta \wedge \Theta(\omega) \notin \text{Activated}(s_2) \wedge \alpha(\omega) \neq 0\}$
- $\text{AP} = V$
- Lab s.t. $\text{Lab}((\rho, \pi)) = \{v \in V \mid v \in \text{Failed}(\pi)\}$

We call $\mathcal{C}_F$ the *computation tree of* F. ■

We do not include transitions which describe component failures whose attached basic event has been triggered before, cf. type (3) in Definition 4.18, as they do not change the internal state of the DFT.

Remark 25. Please notice that we defined Markov automata to not have deadlock states. However, deadlock states can be eliminated as discussed before, so we do define them with deadlocks for simplicity. Moreover, this Markov automaton is indeed an IMC. However, defining it as the more general MA comes at no costs and allows the DFT semantics to be easily extended with PDEPs in Section 4.5 on page 110.

Configuration The locations contain all information required to construct the MA. When introducing the internal state of a DFT, we already saw that we indeed only require the three predicates in that state. We then introduced the triggered events after π, but it is easy to see that these follow directly from the internal state. Likewise, the internal state and the structure of the DFT determine the active components.

Ultimately, we are interested in the properties of the system and not in the internal state of the DFT. The configuration of the system is the information from the DFT that is relevant to construct the computation tree.

Definition 4.20. Given a DFT $F^\Omega = (V, \sigma, \text{Tp}, \Theta, \text{top})$ with the functional transducer $\mathcal{T}_F$. For any state $s = (\rho, \pi)$, the *configuration of* s is the tuple $\mathfrak{C}(s) \in \mathbb{B}^2 \times \Omega^2$ given by

$$(\rho, \pi) \mapsto (\text{top} \in \text{Failed}(\pi), \Delta(\pi) = \emptyset, \{\omega \in \Omega \mid \Theta(\omega) \in \pi\}, \{\omega \in \Omega \mid \Theta(\omega) \in \text{Active}(\pi)\}.$$ ■

The internal state suffices to construct the configuration of a state.

Corollary 4.35. *Let F be a DFT with (ρ, π) and (ρ', π') states in the functional transducer. If $\text{State}_{int}(\pi) = \text{State}_{int}(\pi')$, then $\mathfrak{C}((\pi, \rho)) = \mathfrak{C}((\pi', \rho'))$.*

Reviewing the construction of $\mathcal{T}_F$ and $\mathcal{C}_F$, we see that any model which correctly oracles the configuration for a state yields the same computation tree, up to the action labels and the state labelling.

For the measures of interest, we do not require the full label set and we do not use the action set either. Indeed, we often only require the top node in the label set. Thus, we define $\mathcal{C}_F^*$ as $\mathcal{C}_F$ with the label set restricted to $\{\text{top}\}$.

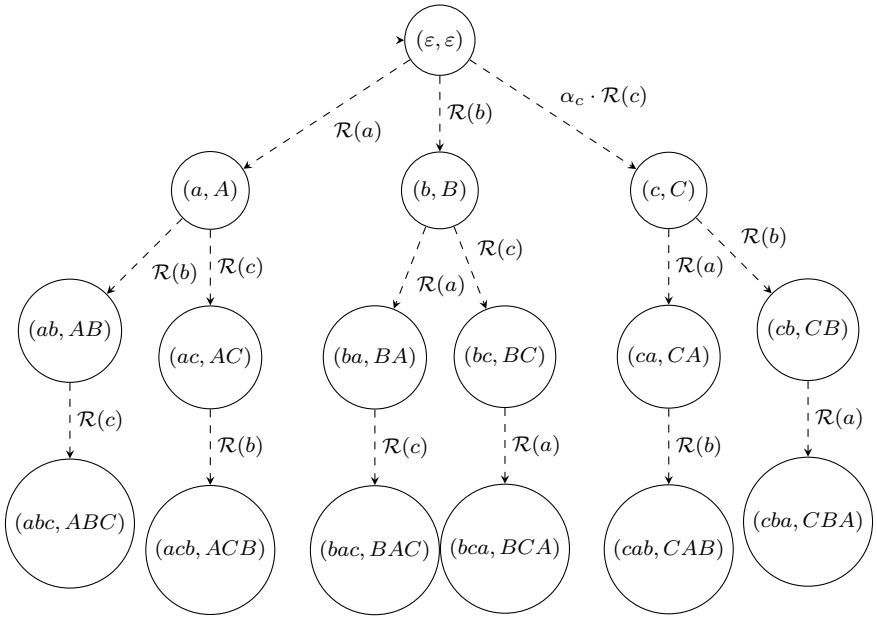


Figure 4.5.: The reachable fragment of the computation tree from Example 4.8.

Examples We give the computation trees for our running examples.

Example 4.8. We consider Example 4.1 on page 75. We describe the computation tree, depicted in Figure 4.5, based on its transducer, cf. Example 4.3 on page 93. The transducer does not contain any ε/x transitions — as there were no FDEPs in the DFT, therefore, the computation tree does not contain immediate transitions. In location $(\varepsilon, \varepsilon)$, the component c is not yet activate, therefore, the rate of the outgoing transition is reduced by the multiplication with the dormancy factor. ▲

Example 4.9. We consider Example 4.2 on page 76. We describe the computation tree, depicted in Figure 4.6, based on its transducer, cf. Example 4.4 on page 93. We see that the reachable state space is smaller. This is due to two reasons. First, the transitions of type (3) are not included. Second, as we are not interested in what happens after the failure of the top level, we omitted all outgoing transitions from locations (ρ, π) with top $\in$ Failed(π). We see that the ε/x transitions are reflected by immediate transitions. As every basic event is in the top-module, all components are active from the start and there is no reduced rate at one of the Markovian transitions. ▲

4.2.8. Policies on DFTs

A policy on DFTs yields for each state of the DFT an ordering of the functional dependencies to be resolved.

Definition 4.21 (DFT Policy)**.** Let F be a DFT. A partial function $\mathsf{Pol} \colon \Omega^{\triangleright} \times \Pi^F \nrightarrow F_{\mathrm{BE}}$ is a *partial policy over* F with

$$\mathsf{Pol}(\rho, \pi) = \begin{cases} \bot & \text{if } \Delta(\pi) = \emptyset \\ X & \text{for some } X \subseteq \Delta(\pi) \end{cases}$$

We call Pol a *(complete) policy over* F if for all ρ, π,

$$\mathsf{Pol}(\rho, \pi) = \bot \vee |\mathsf{Pol}(\rho, \pi)| = 1. \qquad \blacksquare$$

We write $\mathsf{Pol}_\emptyset$ to denote the partial policy over F such that $\mathsf{Pol}(\rho, \pi) = \Delta(\pi)$ for all π with $\Delta(\pi) \neq \emptyset$.

A policy on F restricts the functional transducer of F, by restricting the possible transitions in unresolved states.

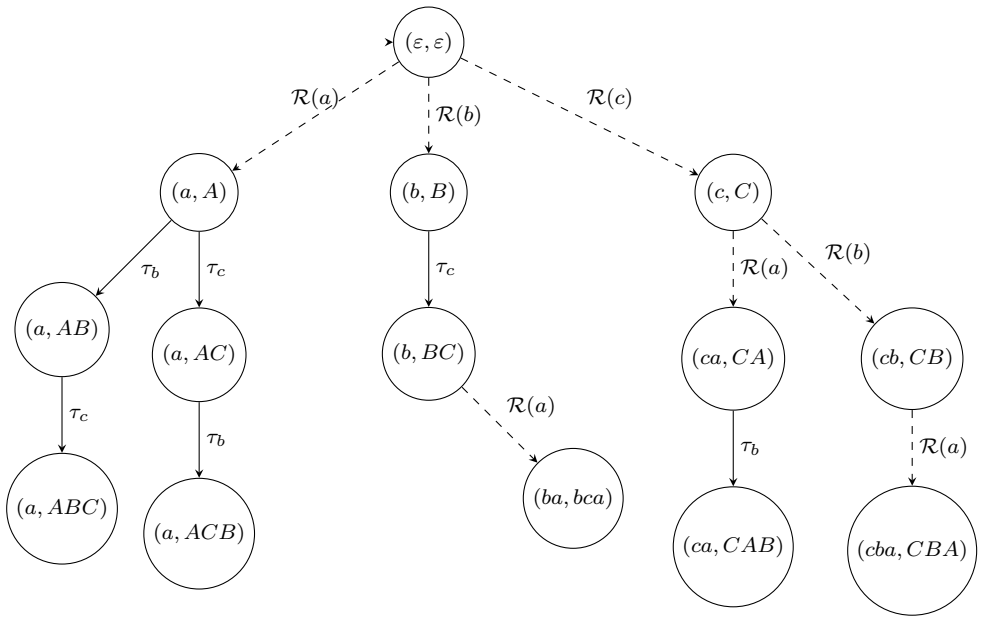


Figure 4.6.: The reachable fragment of the computation tree from Example 4.9.

Definition 4.22 (Policy-induced functional transducer)**.** Given a DFT $F^{\Omega} = (V, E, \mathrm{Tp}, \Theta, \mathrm{top})$ and a policy Pol, we define the transducer $\mathcal{T}[\mathsf{Pol}] = (Q, \Sigma, \Gamma, I, \circledcirc, \delta)$

- State space $Q = \Omega^{\triangleright} \times F_{\mathrm{BE}}^{\triangleright}$,
- Input alphabet $\Sigma = \Omega$.
- Output alphabet $\Gamma = F_{\mathrm{BE}}$.
- Initial states $I = \{(\varepsilon, \varepsilon)\}$.
- Final states $\circledcirc = \{(\rho, \pi) \mid \Delta(\pi) = \emptyset\}$.
- Transition relation $\delta \subseteq Q \times \Sigma \times \Gamma \times Q$, with

$$\begin{aligned}
\delta = &\{((\rho,\pi), \omega, \Theta(\omega), (\rho \cdot \omega, \pi \cdot \Theta(\omega))) && | \omega \notin \rho \wedge \Delta(\pi) = \emptyset \wedge \Theta(\omega) \notin \pi\} \\
&\cup \{((\rho,\pi), \varepsilon, x, (\rho, \pi \cdot x)) && | x \in \mathsf{Pol}(\rho,\pi)\} \\
&\cup \{((\rho,\pi), \omega, \varepsilon, (\rho \cdot \omega, \pi)) && | \omega \notin \rho \wedge \Delta(\pi) = \emptyset \wedge \Theta(\omega) \in \pi\}
\end{aligned}$$

$\mathcal{T}_F[\mathsf{Pol}]$ is called the *functional transducer of F under policy Pol.* ■

The policy-induced computation tree $\mathcal{C}_F[\mathsf{Pol}]$ is defined analogously to the computation tree (Definition 4.19), however on $\mathcal{T}[\mathsf{Pol}]$.

4.2.9. Syntactic sugar

We briefly discuss three extensions[1] to the DFT syntax which do not alter the expressive power of DFTs. Further ideas are discussed in Section 4.5 on page 110.

FDEPs with multiple dependent events We allow FDEPs to have more than one dependent event. An FDEP with multiple dependent events then simply models a set of FDEPs.

Definition 4.23. A (not well-formed) DFT F is called a *DFT with multiple dependent events* if it satisfies all conditions but condition (6) (from the well-formedness definition (Definition 4.9). Instead of condition (6), $\forall v \in F_{\mathrm{FDEP}}.\ |\sigma(v)| = 2$, we require

$$\forall v \in F_{\mathrm{FDEP}}.\ |\sigma(v)| \geq 2.$$ ■

[1] which can be combined, although not formally treated here

We interpret this by simply splitting the FDEPs with multiple dependent events into multiple FDEPs.

Definition 4.24. Given a DFT $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ with multiple dependent events. We define the DFT $F' = (V', \sigma', \mathrm{Tp}', \Theta', \mathrm{top}')$ as equivalent. We define that $\Theta' = \Theta$ and $\mathrm{top} = \top$. Furthermore, let $X = \{v \in F_{\mathrm{FDEP}} \mid |\sigma(v)| > 2\}$ the elements which are removed $Y = \{y_i^x \mid x \in X \wedge 2 \leq i \leq \deg(x)\}$ their replacements. We define

- $V' = V \setminus X \cup Y$.
- $\sigma' = \sigma_{|V \setminus X} \cup \{y_i^x \mapsto \sigma(x)_1 \sigma(x)_i \mid y_i^x \in Y\}$.
- $\mathrm{Tp}' = \mathrm{Tp} \cup \{y \mapsto \mathrm{FDEP} \mid y \in Y\}$. ■

We observe the following (without proof).

Corollary 4.36. *The DFT F' from Definition 4.24 is well-formed.*

Por-gates We extend DFT gate-types with (inclusive) por-gates (POR).

Definition 4.25. An *por-extended DFT* $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ is a regular DFT with $\mathrm{Tp} \to \mathsf{Gates} \cup \mathsf{Leafs}$ replaced by $\mathrm{Tp} \to \mathsf{Gates} \cup \mathsf{Leafs} \cup \{\mathrm{POR}\}$. The por-extended DFT F is well-formed if all regular well-formedness criteria (cf. Definition 4.9) are met and additionally

$$\forall v \in \mathrm{POR}.\ |\sigma(v)| = 2$$

holds. ■

As discussed in Section 3.3.4.6 on page 48, the POR can be modelled using a PAND and an OR.

Definition 4.26. Let $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ be a well-defined por-extended DFT. We define $F' = (V', \sigma', \mathrm{Tp}', \Theta', \mathrm{top}')$ as equivalent. We define

$$Z = \{z_{(x_1 \ldots x_n)}^{(m_1 \ldots m_n)} \in V \mid \forall 1 \leq n.\ m_i \in \mathbb{N} \wedge x_i \in F_{\mathrm{POR}} \wedge x_i = \sigma(z)_{m_i}\}$$

and $Y = \bigcup\{\{y_1^x, y_2^x\} \mid x \in F_{\mathrm{POR}}\}$. Furthermore, we define

- $V' = V \setminus F_{\mathrm{POR}} \cup Y$.
- $$\sigma' = \sigma_{|V' \setminus Z} \cup \{z_{(x_1 \ldots x_n)}^{(m_1 \ldots m_n)} \mapsto v_1 \ldots v_k \mid z_{(x_1 \ldots x_n)}^{(m_1 \ldots m_n)} \in Z \wedge v_i \in V' \text{ with } \}$$
  $$v_i = \begin{cases} y_1^x & \text{if } x = \sigma(z)_i \wedge \exists 1 \leq j \leq n\ m_j = i \\ \sigma(z)_i & \text{otherwise.} \end{cases}$$
  $$\cup \{y_1^x \mapsto \sigma(x)_1 y_2^x \mid y_1^x \in Y\} \cup \{y_2^x \mapsto \sigma(x) \mid y_2^x \in Y\}.$$
- $\mathrm{Tp}' = \mathrm{Tp}_{|V'} \cup \{y_1^x \mapsto \mathrm{PAND} \mid y_1^x \in Z\} \cup \{y_2^x \mapsto \mathrm{OR} \mid y_2^x \in Z\}$.
- $\Theta' = \Theta$
- $\mathrm{top}' = \mathrm{top}$ if $\mathrm{top} \notin F_{\mathrm{POR}}$ and $\mathrm{top}' = z_1^\top$ otherwise. ■

We observe the following (without proof).

Corollary 4.37. *The DFT F' from Definition 4.26 is well-formed.*

Another extension which can be found in e.g. DFTCalc are sequential-and gates (seqand-gates). Seqand-gates only have basic events as successors and none of the successors is allowed to have other predecessors. Seqand-gates fail if all successors have failed, and the basic event are in cold standby until all basic events left of it have failed. They are thus (cold) spare gates with unshared spare modules where each spare module consists of a single basic event.

4.3. Equivalences

We defined a quantitative model for DFTs and showed for some properties that it does indeed fulfil our requirements. In this section, we formally define the most important probabilistic measures for DFTs on this underlying model. Furthermore, we define a notion of equivalency on DFTs, which we use later in Section 4.4 on page 102 and in Chapter 5 on page 113 to prove correctness of rewrite rules.

4.3.1. Quantitative measures on DFTs

We give a formal definition of the probabilistic measures from Section 3.3.3 on page 39.

Definition 4.27 (Failure states)**.** Let F be a DFT and $\mathcal{C}_F = (S, \bar{x}, \iota, A, \hookrightarrow, \dashrightarrow, \mathrm{AP}, L)$ the computation tree of F. Let $\mathsf{Fail}_F = \{s \in S \mid \mathrm{top} \in L(s)\}$. Then Fail_F is the set of *failure states of* F. ■

The probability of failure is the time unbounded probability to reach the set of failure states in the MA.

Definition 4.28 (Minimal probability of failure)**.** Given a DFT F and a partial policy Pol on F, the *minimal probability of failure of* F *under Pol*, $\mathsf{minPrF}_{F[\mathrm{Pol}]}$, is given by

$$\mathsf{minPrF}_{F[\mathrm{Pol}]} = \min_{\mathcal{S} \in \mathrm{DSSched}_{\mathcal{M}}} \Pr_{\mathcal{S}}^{\mathcal{C}_F[\mathrm{Pol}]}(\Diamond \mathsf{Fail}_F). \qquad ■$$

The maximal probability of failure is given analogously.

A DFT is *eventually failing* if the minimal chance of failure is one.

The reliability given a mission time $\mathsf{Rely}_F(t)$ is the time bounded probability to reach the set of failure states in the MA. As for the chance of failure, we define the infimum and the supremum[1] by ranging over all possible schedulers. The (conditional) mean time to failure MTTF_F (CMTTF_F) is the expected time to reach the set of failure states (under the condition that we eventually reach the failure states).

Remark 26. Please notice that the policies induce a restricted class of stationary schedulers. The minimum/maximum values that are realisable for, e.g. reachability, are not necessarily realisable with stationary schedulers and thereby especially not realisable by just ranging over all policies.

We only give the definitions for the minimal values below. In the remainder, we omit the policy whenever we use $\mathrm{Pol}_\emptyset$.

Definition 4.29 (Minimal Reliability)**.** Given a DFT F, a partial policy Pol on F, and a mission time $t \in \mathbb{R}_{>0}$, the *minimal reliability of* F *under Pol given* t, $\mathsf{minRely}_{F[\mathrm{Pol}]}(t)$, is given by

$$\mathsf{minRely}_{F[\mathrm{Pol}]}(t) = 1 - \max_{\mathcal{S} \in \mathrm{DSched}} \Pr_{\mathcal{S}}^{\mathcal{C}_F[\mathrm{Pol}]}(\Diamond^{\leq t} \mathsf{Fail}_F). \qquad ■$$

Definition 4.30 (Minimal Mean time to failure(MTTF))**.** Given a DFT F and a partial policy Pol on F, the *minimal mean time to failure in* F *under Pol* $\mathsf{minMTTF}_{F[\mathrm{Pol}]}$ is given by

$$\mathsf{minMTTF}_{F[\mathrm{Pol}]} \min_{\mathcal{S} \in \mathrm{DSSched}_{\mathcal{M}}} \mathsf{ET}_{\mathcal{S}}^{\mathcal{C}_F[\mathrm{Pol}]}(\Diamond \mathsf{Fail}). \qquad ■$$

Definition 4.31 (Minimal Conditional MTTF)**.** Given a DFT F and a partial policy Pol on F, the *conditional mean time to failure in* F *under Pol*, $\mathsf{minCMTTF}_{F[\mathrm{Pol}]}$ is given by

$$\mathsf{minCMTTF}_{F[\mathrm{Pol}]} \min_{\mathcal{S} \in \mathrm{DSSched}_{\mathcal{M}}} \mathsf{ET}_{\mathcal{S}}^{\mathcal{C}_F[\mathrm{Pol}]}(\Diamond \mathsf{Fail} | \Diamond \mathsf{Fail}). \qquad ■$$

We notice that if the DFT is eventually failing, the MTTF and the CMTTF coincide.

4.3.2. Equivalence classes

We consider equivalence of two DFTs, in order to prepare the development of reduction rules.

Intuitively, two DFTs should be considered equal if they fail after the same set of failure traces. However, such trace equivalences do not preserve the measures we're interested in on the general class of DFTs. Consider the following definition.

[1] We write minimum and maximum, which is fine here as we have only finitely many paths

Definition 4.32 (May and Must fail)**.** Let F be a DFT and $\mathcal{T}$ its functional transducer.

- *May-fail in* F: $\Diamond\mathsf{Fail}_F = \{\rho \mid \exists \pi \in \mathcal{T}(\rho).\ (\rho, \pi) \in \mathsf{Fail}_F\}$
- *Must-fail in* F: $\Box\mathsf{Fail}_F = \{\rho \mid \forall \pi \in \mathcal{T}(\rho).\ (\rho, \pi) \in \mathsf{Fail}_F\}$ ■

DFTs with equivalent may-fail and must-fail sets do not have the same MTTF and reliability as there are two orthogonal problems.

1. *The moment of activation.* The naive encoding in may-fail and must-fail does not account for different points of changing from a cold to a hot failure rate. This may yield different probability masses for the paths in $\Diamond\mathsf{Fail}_F$ ($\Box\mathsf{Fail}_F$) and $\Diamond\mathsf{Fail}_{F'}$ ($\Box\mathsf{Fail}_{F'}$). Please notice that one could easily extend the notion of these paths to include information about the set of activated components.

2. *The moment of choice.* Like all trace equivalences, the moments where non-determinism is resolved is not taken into account. Whereas we're not really interested in the moment or the way the non-determinism is resolved, this directly affects the time-bounded properties and thus the measures we want to preserve. It is important to notice that restricting ourselves to time-independent schedulers does not resolve this issue, as the foundation of this problem lies in the fact that the non-determinism may occur before any basic event has occurred, or after a non-empty failure trace.

Notice that on DFTs without non-determinism or warm components, this notion above indeed preserves the measures-of-interest.

For the general class of DFTs, we confine ourselves to a more conservative equivalence criterion. A simple approach is to only define the equivalence on the underlying computation trees.

Strong equivalence

Definition 4.33 (Strongly equivalent)**.** Given two DFTs F, F'. The DFTs F and F' are *strongly equivalent*, denoted $F \equiv F'$, whenever $\mathcal{C}^*_F \approx_w \mathcal{C}^*_{F'}$. ■

We would like to argue on a qualitative level. Using the notions from Section 4.2.7 on page 96, we introduce the following equivalence relation.

Definition 4.34. Let F be a DFT with $\mathcal{T}_F = (Q, \Sigma, \Gamma, I, \circledcirc, \delta)$. Let $R \subseteq Q \times Q$ be an equivalence relation with $(s, s') \in R$. If

- $\mathfrak{C}(s) = \mathfrak{C}(s')$
  $\forall t \in Q \exists \omega \in \Omega \cup \{\varepsilon\}\ e \in F_{\text{BE}} \cup \{\varepsilon\}\ (s, \omega, e, t)$
- $\implies$
  $\exists t' \in Q \exists e' \in F_{\text{BE}} \cup \{\varepsilon\}\ (s', \omega, e', t') \in \delta \wedge (t, t') \in R$

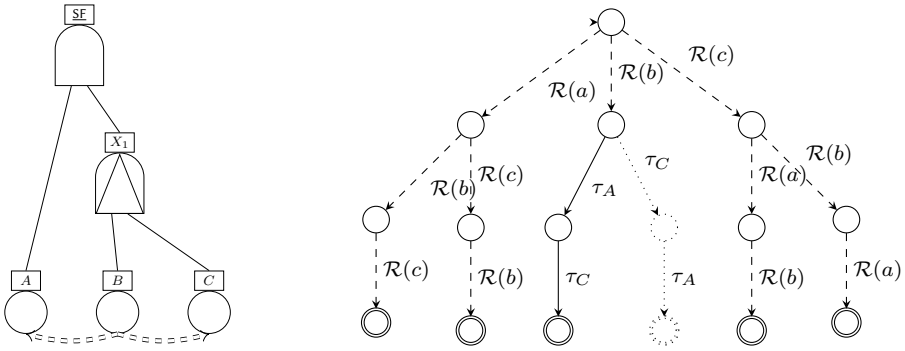
then we call R a configuration-equivalence. Two states $s, s' \in Q$ are *configuration-equivalent* if there exists a configuration-equivalence R such that $(s, s') \in R$. Two DFTs F, F' are configuration-equivalent if in the disjoint union of $\mathcal{T}_F$ and $\mathcal{T}_{F'}$ their initial states are configuration equivalent. ■

We directly obtain that configuration-equivalence suffices to imply strong equivalence.

Corollary 4.38. *Given two DFT F, F'. If F and F' are configuration-equivalent, then they are strongly equivalent.*

For abstraction of a single DFT, the following notion, which follows from Corollary 4.35 is helpful.

Corollary 4.39. *Given a DFT F with the functional transducer $\mathcal{T}_F$ and two states $(\rho, \pi), (\rho, \pi')$ in the functional transducer. If $\mathit{State}_{\mathit{int}}(\pi) = \mathit{State}_{\mathit{int}}(\pi')$ then (ρ, π) and (ρ, π') are configuration-equivalent.*



(a) The DFT.

(b) The corresponding functional transducer.

Figure 4.7.: A positive example for partial order reduction.

Weak equivalence In Chapter 5 on page 113, we want to eliminate subtrees which are infallible. In particular, we want to remove dispensable basic events. However, this would not be covered be the notion above. We therefore define the very liberal notion of equivalence based on the measures defined above.

Definition 4.35 (Weakly equivalent)**.** Given two DFTs F, F'. The DFTs F and F' are weakly equivalent, denoted $F \approxeq F'$, whenever all of the following conditions hold.

- $\mathsf{minPrF}_F = \mathsf{minPrF}_{F'}$ and $\mathsf{maxPrF}_F = \mathsf{maxPrF}_{F'}$.
- $\forall t \in \mathbb{R}_{\geq 0}.\ \mathsf{minRely}_F(t) = \mathsf{minRely}_{F'}(t)$ and $\forall t \in \mathbb{R}_{\geq 0}.\ \mathsf{maxRely}_F(t) = \mathsf{maxRely}_{F'}(t)$.
- $\mathsf{minMTTF}_F = \mathsf{minMTTF}_{F'}$ and $\mathsf{maxMTTF}_F = \mathsf{maxMTTF}_{F'}$. ■

First, strong equivalence implies weak equivalence, cf. Section 2.2.3 on page 15.

Corollary 4.40. *Let F, F' be two DFTs. If $F \equiv F'$ then also $F \approxeq F'$.*

Second, a DFT F with unconnected basic events is indeed weakly equivalent to a DFT with this BE removed. We discuss this in detail in Section 5.2.4 on page 128

We notice that we always have to argue about all policies. In the next section, we discuss how we can restrict the policies under consideration

4.4. Partial order reduction for DFTs

Partial order reduction (POred) is a well known technique in model checking [BK08]. It aims to reduce non-determinism due to interleaving, as it introduces a fixed ordering on sets of interleaving events for which the actual ordering is unknown. To preserve semantics, it is important that the outcome of such a set of events doesn't rely on the ordering applied on it.

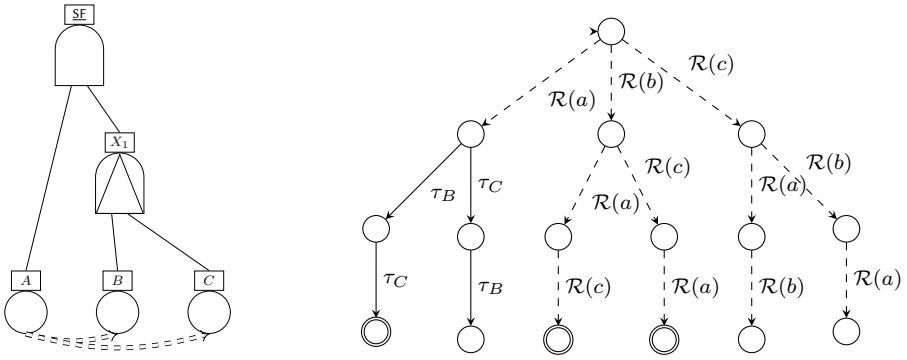
We introduce partial order reduction here for two reasons. First, we have seen that a large part of the non-determinism in DFTs is pointless, i.e., for each scheduler we get the same result. Applying POred might give a speed up in the state-space generation, which is clearly a bottleneck in existing tool sets.

Second, and more important in our context, the notion of POred simplifies correctness proofs of rewrite rules which include FDEP-gates. We illustrate the idea with two small examples - one in which the outcome does not rely on the ordering and one in which it does.

Example 4.10. We consider the DFT depicted in Figure 4.7a. After the failure of B, both A and C are triggered. The order in which A and C then fail does not matter, even in the presence of the PAND. In Figure 4.7b, we depict the functional transducer. If we assume that A is always triggered before C is, then the dotted part of the functional transducer is eliminated. Notice that the resulting functional transducer is then deterministic. ▲

This ordering obviously does not work if it affects both children of the PAND.

Example 4.11. We consider the DFT depicted in Figure 4.8a. After the failure of A, both B and C are triggered. The order in which B and C then fail matters, as first B and then C causes the



(a) The DFT.

(b) The corresponding functional transducer.

Figure 4.8.: A negative example for partial order reduction.

system to fail, while first triggering C and then B renders the PAND infallible. Therefore, we cannot make the functional transducer (Figure 4.8b) deterministic. ▲

It is not surprising that the partial order reduction is not valid in the latter example. Like the *ample-set construction* [BK08], we now want simple rules on the level of DFTs which allow us to omit parts of the underlying transducer. We first require some further terminology for basic events and their connection via FDEPs.

Definition 4.36 (Potential enabling)**.** Given a DFT $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$, for each $v \in F_{\mathrm{BE}}$, we define the *potential triggered FDEPs* as $\mathrm{PotTrig}(v) = \{v' \in F_{\mathrm{FDEP}} \mid \exists p \in \mathrm{ifcp}(v')\ p_\downarrow = v\}$. Furthermore, we define for $v \in \mathrm{FDEP}$, the set $\mathrm{PotEnables}(v)$ as the set of functional dependencies which might be triggered after v is triggered, that is, $\mathrm{PotEnables}(v) = \mathrm{PotTrig}(v_2)$. ■

We call basic events where different orders in which they fail yield *in conflict*. We give a very naive attempt to characterise events that are in conflict.

Remark 27. We call this naive as this version indeed reflects an idea which emerged very early in the preparation for this thesis. Later on, it became clear that some other events are also in conflict. However, as we will see, this naive concept is already very powerful and for a large class of DFTs, it is powerful enough.

Definition 4.37 (Naive mutual conflict)**.** Let $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ be a DFT with $x, y \in F_{\mathrm{BE}}$.
We call v and v' in *mutual pand-conflict* if there exists a pand-gate $z \in F_{\mathrm{PAND}}$ such that

$$\exists i, j < \deg(z)\ i \neq j \wedge \exists p \in \mathrm{ifcp}(\sigma(z)_i)\ p_\downarrow = x \wedge \exists p' \in \mathrm{ifcp}(\sigma(z)_j)\ p'_\downarrow = y.$$

Furthermore, v and v' are in *naive mutual spare-conflict* if there exist two different spare gates $s, s' \in F_{\mathrm{SPARE}}$, s.t.

$$\exists z \in \sigma(s) \exists z' \in \sigma(s') \exists p = \mathrm{ifcp}(\sigma(z))\ p_\downarrow = x \wedge \exists p' \in \mathrm{ifcp}(\sigma(z'))\ p'_\downarrow = y$$

We call v and v' in *naive mutual conflict*, if they're either in mutual pand-conflict or in naive mutual spare-conflict. Otherwise, they're *naively conflict-free*. ■

Please, notice that the notions are over-approximations. There are numerous cases which are considered conflicting here but are not problematic in practice, e.g. spares in independent subtrees may fail in any order, as they never claim the same successors.

We have given a syntactical meaning for being naively conflict-free. The next proposition yields a semantic consequence.

Proposition 4.41. *Let F^Ω be a DFT and $\mathcal{T}_F$ its functional transducer with a state $(\rho, \pi) \in \Omega^\triangleright \times F_{BE}^\triangleright$ and $\{x, y\} \subseteq \Delta(\pi)$ with x, y naively conflict-free. For $(\rho, \pi \cdot xy)$ and $(\rho, \pi \cdot yx)$ it holds that Failed$(\pi \cdot xy) =$ Failed$(\pi \cdot yx)$.*

Proof. Let $C = \mathrm{Failed}(\pi \cdot xy)$, and $C' = \mathrm{Failed}(\pi \cdot yx)$. We first show $C = C'$. We use a structural induction. Assume that not, than w.l.o.g. $\exists v \in C \setminus C'$

$v \notin F_{\text{BE}}$ as $v \in \{e \in \pi\} \cup \{x,y\}$.

$v \in F_{\text{VOT}}(k)$ then there exists some child $v' \in C \setminus C'$ which contradicts the induction hypothesis.

$v \in F_{\text{PAND}}$.

$$
\begin{aligned}
& v \in C \setminus C' && \Longrightarrow \\
& v \in \text{Failed}(\pi \cdot xy) \wedge v \notin \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& \quad \sigma(v) \subseteq \text{Failed}(\pi \cdot xy) \wedge v \in \text{Failable}(\pi \cdot x) \wedge v \notin \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& \sigma(v) \subseteq \text{Failed}(\pi \cdot xy) \wedge v \in \text{Failable}(\pi \cdot x) \wedge \\
& (\sigma(v) \not\subseteq \text{Failed}(\pi \cdot xy) \vee v \notin \text{Failable}(\pi \cdot y)) && \Longrightarrow \\
& \quad (\sigma(v) \subseteq \text{Failed}(\pi \cdot xy) \wedge v \in \text{Failable}(\pi \cdot x) \wedge \sigma(v) \not\subseteq \text{Failed}(\pi \cdot yx)) \vee \\
& (\sigma(v) \subseteq \text{Failed}(\pi \cdot xy) \wedge v \in \text{Failable}(\pi \cdot x) \wedge \\
& \quad v \notin \text{Failable}(\pi \cdot y) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx)) && \Longrightarrow \\
& \exists v' \in \sigma(v).\ v' \in C \setminus C' \vee \\
& (\sigma(v) \subseteq \text{Failed}(\pi \cdot xy) \wedge v \in \text{Failable}(\pi \cdot x) \wedge \\
& \quad v \notin \text{Failable}(\pi \cdot y) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx)) && \Longrightarrow \\
& \sigma(v) \subseteq \text{Failed}(\pi \cdot xy) \wedge v \in \text{Failable}(\pi \cdot x) \wedge \\
& \quad v \notin \text{Failable}(\pi \cdot y) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& v \in \text{Failable}(\pi) \wedge v \notin \text{Failable}(\pi \cdot y) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& v \notin \text{Failable}(\pi \cdot y) \wedge v \in \text{Failable}(\pi) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& (v \notin \text{Failable}(\pi) \vee \\
& \quad (\exists j < \deg(v).\ \sigma(v)_{j+1} \in \text{JustFailed}(\pi \cdot y) \wedge \sigma(v)_j \notin \text{Failed}(\pi \cdot y))) \wedge \\
& v \in \text{Failable}(\pi) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& \exists j < \deg(v).\ \sigma(v)_{j+1} \in \text{JustFailed}(\pi \cdot y) \wedge \\
& \quad \sigma(v)_j \notin \text{Failed}(\pi \cdot y) \wedge \sigma(v) \subseteq \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& \exists j < \deg(v).\ \sigma(v)_j \in \text{JustFailed}(\pi \cdot yx) \wedge \sigma(v)_{j+1} \in \text{JustFailed}(\pi \cdot y) && \Longrightarrow \\
& \exists j < \deg(v).\ x \in \text{ifcp}(\sigma(v)_j) \wedge y \in \text{ifcp}(\sigma(v)_{j+1})
\end{aligned}
$$

Which contradicts the fact that x and y are not in mutual conflict.

$v \in F_{\text{SPARE}}$

Let $z = \text{LastClaimed}(\pi \cdot x, v) = \text{LastClaimed}(\pi \cdot y)$. We can conclude $z \notin \text{Failed}(\pi)$ and

$$z \in \text{Failed}(\pi \cdot xy) \wedge z \in \text{Failed}(\pi \cdot yx),$$

otherwise, $\exists v' \in \sigma(v).\ v' \in (C \setminus C')$ which contradicts the induction hypothesis.

Let us first assume $v \in \text{JustFailed}(\pi \cdot x)$. We notice that this should lead to a contradiction without resorting to the conflict-free graph. To see why, we see that the failure of x suffices to let v fail, that is, v is not able to claim anything anymore even before the failure of x and x causes the last claimed successor z to fail. In order to ensure that v does not fail after yx, the failure of y would need to prevent the failure of z, but that is excluded by the induction hypothesis. We formalise this below.

As argued above, the following suffices to come to a contradiction.

$$
\begin{aligned}
& v \in \text{JustFailed}(\pi \cdot x) && \Longrightarrow \\
& z \in \text{Failed}(\pi \cdot x) \wedge \text{Available}(\pi \cdot x, v) = \emptyset && \Longrightarrow \\
& z \in \text{Failed}(\pi \cdot x) \wedge \forall z' \in \sigma(v).\ \text{ClaimedBy}(\pi, z') \neq \emptyset \vee z' \in \text{Failed}(\pi \cdot x)
\end{aligned}
$$

We use this later in the following argument.

$$
\begin{aligned}
& z \in \text{Failed}(\pi \cdot yx) && \Longrightarrow \\
& z \in \text{JustFailed}(\pi \cdot y) \vee (\text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx)) && \Longrightarrow \\
& v \in \text{Failed}(\pi \cdot yx) \vee \text{Available}(\pi \cdot y, v) \neq \emptyset \vee \\
& \quad (\text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx)) && \Longrightarrow \\
& \text{Available}(\pi \cdot y, v) \neq \emptyset \vee (\text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx)) && \Longrightarrow \\
& (\exists z' \in \sigma(v).\ \text{ClaimedBy}(\pi, z') = \emptyset \wedge z' \notin \text{Failed}(\pi \cdot y)) \vee \\
& \quad (\text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx)) && \Longrightarrow \\
& (\exists z' \in \sigma(v).\ \text{ClaimedBy}(\pi, z') = \emptyset \wedge z' \notin \text{Failed}(\pi \cdot y) \wedge z' \in \text{Failed}(\pi \cdot x)) \vee \\
& \quad (\text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx)) && \Longrightarrow \\
& (\exists z' \in \sigma(v).\ \text{ClaimedBy}(\pi, z') = \emptyset \wedge z' \in \text{JustFailed}(\pi \cdot yx) \wedge \\
& \quad z' \in \text{JustFailed}(\pi \cdot x)) \vee \\
& \quad (\text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx)) && \Longrightarrow \\
& \text{LastClaimed}(\pi \cdot y, v) = z \wedge z \in \text{JustFailed}(\pi \cdot yx) && \Longrightarrow \\
& (z \notin \text{Failed}(\pi \cdot y) \wedge \text{Available}(\pi \cdot yx, v) \neq \emptyset \vee v \in \text{Failed}(\pi \cdot yx))) && \Longrightarrow \\
& (z \notin \text{Failed}(\pi \cdot y) \wedge (\texttt{false}) && (*)
\end{aligned}
$$

We conclude that $v \in \text{JustFailed}(\pi \cdot xy)$. Therefore, we have that

$$\exists v' \in \sigma(v)\ \exists p \in \text{ifcp}(v')\ p_{\downarrow} = y$$

First, we apply some definitions to come to a useful case distinction.

$$
\begin{aligned}
& v \in \text{JustFailed}(\pi \cdot xy) && \Longrightarrow \\
& v \notin \text{Failed}(\pi \cdot x) \wedge \text{LastClaimed}(\pi \cdot x) \in \text{Failed}(\pi \cdot xy) \wedge \text{Available}(\pi \cdot xy, v) = \emptyset && \Longrightarrow \\
& (\text{LastClaimed}(\pi)(v) \notin \text{Failed}(\pi \cdot x) \vee \text{Available}(\pi \cdot x, v) \neq \emptyset) \wedge \\
& \quad \text{LastClaimed}(\pi \cdot x)(v) \in \text{Failed}(\pi \cdot xy) \wedge \text{Available}(\pi \cdot xy, v) = \emptyset && \Longrightarrow \\
& (\text{LastClaimed}(\pi)(v) \notin \text{Failed}(\pi \cdot x) \wedge \text{LastClaimed}(\pi \cdot x)(v) \in \text{Failed}(\pi \cdot xy) \wedge \\
& \quad \text{Available}(\pi \cdot xy, v) = \emptyset) \vee \\
& \quad (\text{Available}(\pi \cdot x, v) \neq \emptyset \wedge \text{LastClaimed}(\pi \cdot x)(v) \in \text{Failed}(\pi \cdot xy) \wedge \\
& \quad \text{Available}(\pi \cdot xy, v) = \emptyset
\end{aligned}
$$

We see two cases. For the first case, we add $\sigma(v) \cap \text{Available}(\pi \cdot x) = \emptyset$, otherwise, the second case below does apply. The first case is already contradictory without resorting to assumption that x and y are conflict-free.

$$
\begin{aligned}
& \text{LastClaimed}(\pi)(v) \notin \text{Failed}(\pi \cdot x) \wedge \text{LastClaimed}(\pi \cdot x)(v) \in \text{Failed}(\pi \cdot xy) \wedge \\
& \text{Available}(\pi \cdot xy, v) = \emptyset \wedge \text{Available}(\pi \cdot x) = \emptyset && \Longrightarrow \\
& \text{Available}(\pi \cdot x, v) = \emptyset
\end{aligned}
$$

Which yields a contradiction as in (*). We thus conclude

$$\text{Available}(\pi \cdot x, v) \neq \emptyset \wedge \text{LastClaimed}(\pi \cdot x)(v) \in \text{Failed}(\pi \cdot xy) \wedge \sigma(v) \cap \text{Available}(\pi \cdot xy) = \emptyset$$

Now, we use that $v \notin \text{Failed}(\pi \cdot yx)$. We have that:

$$v \notin \text{Failed}(\pi \cdot yx) \implies \text{LastClaimed}(\pi \cdot y, v) \notin \text{Failed}(\pi \cdot yx) \vee \text{Available}(\pi \cdot yx, v) \neq \emptyset$$

We show that in both cases, we can deduce that there exists an $s \in F_{\text{SPARE}} \setminus v$ such that s is claiming after $\pi \cdot x$.

- We start with the latter case, i.e. we assume $\text{Available}(\pi \cdot yx) \neq \emptyset$. Thus, $\exists z' \in \sigma(v)\ z' \notin$

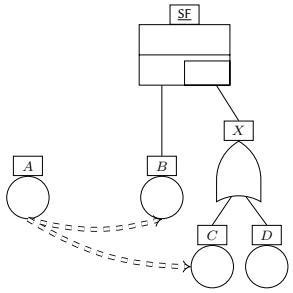


Figure 4.9.: Naive conflict-free events and a single spare-gate.

Failed$(\pi \cdot yx) \wedge$ ClaimedBy$(\pi \cdot y, z') = \emptyset$. By the induction hypothesis, $z' \notin$ Failed$(\pi \cdot xy)$. Therefore, ClaimedBy$(\pi \cdot x, z') = \{s\}$ for some $s \in F_{\text{SPARE}}$. As we have ClaimedBy$(\pi \cdot y, z') = \emptyset$, we can conclude that s claims z' after $\pi \cdot x$. In particular, LastClaimed$(\pi \cdot x, s) = z'$.

We show that $s \neq v$. Assume that $s = v$, we deduce that $z' =$ LastClaimed$(\pi \cdot x, v) \in$ Failed$(\pi \cdot xy)$ and that $z' \notin$ Failed$(\pi \cdot xy)$.

- We now discuss the former case, i.e. we assume LastClaimed$(\pi \cdot y, v) \notin$ Failed$(\pi \cdot yx)$. Let $z' =$ LastClaimed$(\pi \cdot y, v)$. By the induction hypothesis, we have $z' \notin$ Failed$(\pi \cdot xy)$. Furthermore, we have that $z' \notin$ Available$(\pi \cdot xy)$. It follows that for some $s \in F_{\text{SPARE}}$ (with $s \neq v$ as above).

  $$\text{ClaimedBy}(\pi \cdot x, z') = \{s\} \wedge \text{ClaimedBy}(\pi \cdot x, z') = \{v\}$$

  It follows that $z' \neq z$. Notice that s claiming z' after π leads directly to a contradiction with ClaimedBy$(\pi \cdot x, z') = \{v\}$. Therefore, s claims z' after $\pi \cdot x$.

As $s \neq v$ claiming after $\pi \cdot x$, LastClaimed$(\pi, s) \in$ JustFailed$(\pi \cdot x)$ and therefore,

$$\exists j < \deg(s) \exists p \in \text{ifcp}(\sigma(s)_j)\ p_{\downarrow} x)$$

We already had

$$\exists v' \in \sigma(v)\ \exists p \in \text{ifcp}(v') p_{\downarrow} = y$$

which contradicts the conflict-freeness of x and y.

□

The following corollary follows directly.

Corollary 4.42. *Let F^{Ω} be a DFT and $\mathcal{T}_F$ its functional transducer with a state $(\rho, \pi) \in \Omega^{\triangleright} \times F_{BE}^{\triangleright}$ and $\{x, y\} \subseteq \Delta(\pi)$ with x, y naively conflict-free. For $(\rho, \pi \cdot xy)$ and $(\rho, \pi \cdot yx)$ it holds that Failable$(\pi \cdot xy) =$ Failable$(\pi \cdot yx)$.*

The naive version of mutual conflict-free is not strong enough to yield equivalent claiming, which we illustrate in the following example.

Example 4.12. We consider the DFT depicted in Figure 4.9. We consider the initial failure of A. The failure triggers B and C. We see that independent of the order of B and C, as soon as they've both failed then the spare-gate fails. If C is triggered first, then neither C nor D are ever activated. That is, the failure rate of D is reduced, and thereby the probability that D triggers X is reduced.s Notice that the problem here is thus that the activation of the spare module is important even though the module representant fails immediately afterwards. ▲

Based on the example above, we consider more events to be in mutual conflict, and thereby weaken the concept of conflict-free.

Definition 4.38 (Activation-sensitive mutual conflict)**.** Let $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$ be a DFT with $x, y \in F_{\text{BE}}$. We say that v and v' are in *(activation-sensitive) mutual spare-conflict* if there exist two different spare gates $s, s' \in F_{\text{SPARE}}$, s.t.

$$\exists z \in \sigma(s) \exists z' \in \sigma(s') \exists p = \text{ifcp}(\sigma(z))\ p_{\downarrow} = x \wedge \exists p' \in \text{ifcp}(\sigma(z'))\ p'_{\downarrow} = y$$

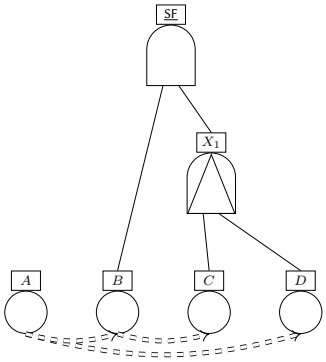


Figure 4.10.: Other dependent events

We call v and v' in *(activation-sensitive) mutual conflict*, if they're either in mutual pand-conflict or in mutual spare-conflict. ■

We call two events *confict-free* if they're not in activation-sensitive mutual conflict.

Proposition 4.43. *Let F^{Ω} be a DFT and $\mathcal{T}_F$ its functional transducer with a state $(\rho, \pi) \in \Omega^{\triangleright} \times F_{BE}^{\triangleright}$ and $\{x, y\} \subseteq \Delta(\pi)$ with x, y naively conflict-free. For $(\rho, \pi \cdot xy)$ and $(\rho, \pi \cdot yx)$ it holds that ClaimedBy$(\pi \cdot xy) =$ ClaimedBy$(\pi \cdot yx)$.*

We omit the proof here due to limited space. The proof follows the proof for Proposition 4.41. Notice that we may use that Failed$(\pi \cdot xy) =$ Failed$(\pi \cdot yx)$, as activion-sensitive conflict-free implies naive conflict-free. This also leads to the following important corollary.

Corollary 4.44. *Let F^{Ω} be a DFT and $\mathcal{T}_F$ its functional transducer with a state $(\rho, \pi) \in \Omega^{\triangleright} \times F_{BE}^{\triangleright}$ and $\{x, y\} \subseteq \Delta(\pi)$ with x, y naively conflict-free. It holds that*

$$\mathfrak{C}(\rho, \pi \cdot xy) = \mathfrak{C}(\rho, \pi \cdot yx).$$

Until now, we only considered two basic events and assumed that they were consequetively triggered. In the presence of other dependent events, we cannot simply assume that the outcome is order independent.

Example 4.13. We consider the DFT depicted in Figure 4.10. A failure of A triggers B and D, which are conflict-free. If we assume that D is triggered before B, then the DFT never fails (the event chain is $ADBC$). On the other hand, if we assume that B fails first, we have the conflicting dependent events C and D left. The path $ABCD$ causes the DFT to fail. ▲

We generalize the notion of conflict-free to commutativity.

Definition 4.39. Let $R \subseteq F_{\text{BE}} \times F_{\text{BE}}$ be the largest relation such that for all vRv'

- v, v' are conflict-free, and
- for all $w' \in \text{PotEnables}(v')$, vRw', and
- for all $w \in \text{PotEnables}(v)$, wRv'.

For any vRv', we say that v and v' are mutual commutative. We denote R with $\circledast$. ■

Remark 28. The notion of commutativity is symmetric, but it is not transitive.

This notion indeed suffices to tackle

Lemma 4.45. *Let F be a DFT and $\mathcal{T}$ its functional transducer. Let $s = (\rho, \pi)$ denote a state in $\mathcal{T}$. Let $x \in \Delta(\pi)$ such that $x \circledast y$ for all $x \neq y \in \Delta(\pi)$. Let $sx = (\rho, \pi \cdot x)$. Let $\tau(s)$ denote the states reachable from s by ε,y transitions. Then*

$$\{\mathfrak{C}(\hat{s}) \mid \hat{s} \in \tau(s) \cap \circledcirc\} = \{\mathfrak{C}(\hat{s}) \mid \hat{s} \in \tau(sx) \cap \circledcirc\}$$

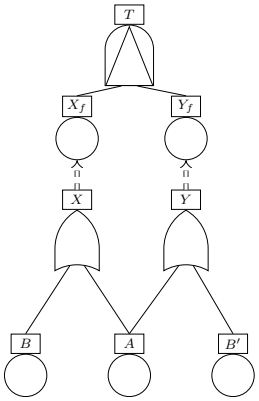


Figure 4.11.: FDEPs being triggered simultaneously

.

We omit some technical details, the proof goes along many proofs for (bi)simulation equivalence and partial order reductions, see [BK08].

Proof sketch. W.l.o.g. we can assume $\Delta(\pi) > 1$. Let $sy = (\rho, \pi \cdot y)$ for some $x \neq y \in \Delta(\pi)$. Let $\mathfrak{C}(\tau(s))$ denote $\{\mathfrak{C}(\hat{s}) \mid \hat{s} \in \tau(s) \cap \circledcirc\}$. It suffices to show that

$$\mathfrak{C}(\tau(sy)) = \mathfrak{C}(\tau(sx))$$

Assume that this is not the case, then there exists an $\hat{sy} \in \mathfrak{C}(\tau(sy)) \setminus \mathfrak{C}(\tau(sx))$ — the other direction is analogous.

Let

$$s \xrightarrow{y} s_1 \xrightarrow{z_1} s_2 \to \ldots \hat{sy}$$

denote this corresponding path — notice that $x = z_i$ for some i. We show that any such path can be mimicked by starting with

$$s \xrightarrow{x} s'_0 \ldots$$

From s'_0, we continue with $\xrightarrow{y}$ to s'_1. This is certainly possible as $y \in \Delta(\pi)$. We notice that $\mathfrak{C}(s'_1) = \mathfrak{C}(t)$ with $s_1 \cdot (\varepsilon, x)$. Either $z_1 = x$, and we are done. Otherwise, we repeat this construction. Notice that $z_1 \in \Delta(\pi \cdot xy)$ as we can conclude from $z_1 \in \Delta(\pi \cdot y)$ that $z_1 \in \Delta(\pi \cdot yx)$ and from there it follows with configuration equivalence. □

The lemma contains explicitely states that one has to consider all dependent events. In particular, this also includes events which are triggered via another path.

Example 4.14. We consider the DFT in Figure 4.11. The functional dependencies triggered by X and Y are conflicting. We notice that they may be triggered at the same time by the failure of A. ▲

Along the lines of potentially triggered, we can now define which dependencies should be considered to always fulfil the requirements of Lemma 4.45. We state the following without proof.

Proposition 4.46. *Let F be a DFT and $\pi \cdot \pi' \in F^{\triangleright}_{BE}$ with $\Delta(\pi) = \emptyset$. Then*

$$\Delta(\pi \cdot \pi') \subseteq \bigcup_{x \in \pi'} \mathit{PotEnables}(x)$$

.

We can use this to proposition to syntacticly determine a superset of the basic events which might be triggered concurrently with a given basic element.

Corollary 4.47. *Let F be a DFT with $x \in F_{BE}$. It holds that*

$$x \in \Delta(\pi) \implies \Delta(\pi) \subseteq \{y \in F_{BE} \mid \exists z \in F_{BE}\, x \in \mathit{PotEnables}^*(z) \wedge y \in \mathit{PotEnables}^*(z)\},$$

where PotEnables* denotes the transitive closure of PotEnables.

The transitive closure of PotEnables captures the set of all elements which are on a *delayed* failure path, which we do not formally introduce here. Intuitively, a delayed failure path is a failure path which might lead through FDEPs.

Based upon merging the corollary with Lemma 4.45 and Corollary 4.38, we arrive at the following theorem.

Theorem 4.48. *Let F_Ω be a DFT and $x \in F_{BE}$ such that $x \circledast y$ for all $y \in \{y \in F_{BE} \mid \exists z \in F_{BE}\, x \in \mathit{PotEnables}^*(z) \wedge y \in \mathit{PotEnables}^*(z)\}$. Let Pol be a partial policy such that $\mathit{Pol}(\rho, \pi) = \{x\}$ if $x \in \Delta(\pi)$. The computation tree $\mathcal{C}_F[\mathit{Pol}] \approx \mathcal{C}_F$. We call x* preferential.

Again, we omit technical details in the proof.

Proof sketch. For any location (ρ, π) with $x \in \Delta(\pi)$, we have by Lemma 4.45 that all $\forall s \in \tau(\rho, \pi) \exists s' \in \tau(\rho, \pi \cdot x)$ such $\mathfrak{C}(s) = \mathfrak{C}(s')$. Therefore, $s \approx_s s'$ by Corollary 4.38. The by τ-transition reachable equivalence classes w.r.t. strong bisimulation are thus identical for (ρ, π) and $(\rho, \pi \cdot x)$ and neither of the states is accepting. Thus (ρ, π) and $(\rho, \pi \cdot x)$ are weakly bisimilar. The bisimulation quotients of $\mathcal{C}_F$ and $\mathcal{C}_F$ are isomorphic, as the restriction to Pol thus only removes transitions inside equivalence classes. □

The theorem thus allows us to reduce the non-determinism introduced by multiple FDEPs, as we can assume a total ordering on the independent FDEPs.

Often, the delayed failure forwarding through dependent events behaves exactly as the immediate failure propagation. We call this phenomenon *δ-independence*. δ-independence can be characterised in a very similar fashion to preferential basic events. We therefore restrict ourselves to a brief overview.

We use δ-independence to eliminate the gap between a basic event x being caused by a component failure and a dependent event y. We assume y to be preferential here to ease the definitions. We then define the δ-independence as a property of the tuple, and we can assume that x occurs before y in the setting we're interested in.

We use the following syntactic criterion to find independent tuples:

Definition 4.40. Let F be a DFT with $x, y \in F_{\mathrm{BE}}$. We say that y is *ignorant about* x if

$$\begin{aligned}
&\forall v \in F_{\mathrm{SPARE}} \forall p \in \mathrm{ifcp}(v).\ p_\downarrow \neq x \wedge \\
&\forall v \in F_{\mathrm{PAND}} i = \min\{v_i \in \sigma(v) \mid \exists p \in \mathrm{ifcp}(v_i).\ p_\downarrow = y\} \\
&\quad \forall j > i \forall p' \in \mathrm{ifcp}(\sigma(v)_j) p'_\downarrow \neq x
\end{aligned}$$

■

Remark 29. Notice that the ignorance is not symmetric. However, we could likewise define ignorance from x to y.

Based on this syntactic assumption, we can then show the following semantic consequence, which we not prove here.

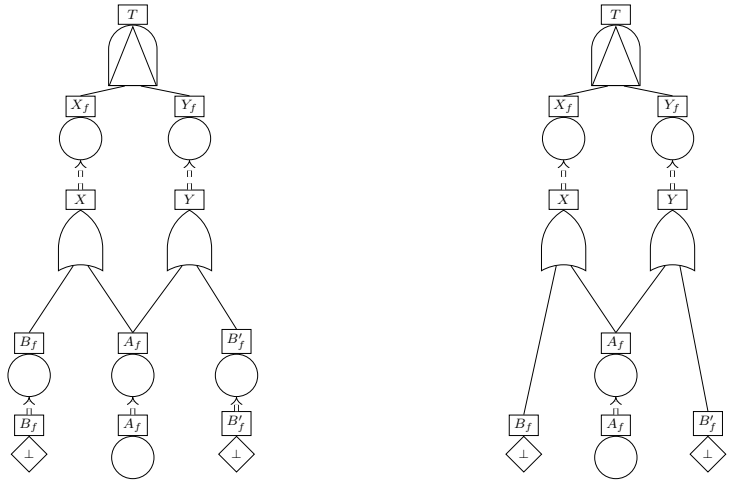
Proposition 4.49. *Let F be a DFT with $\{x, y\} \subseteq F_{BE}$ and $\pi \in F_{BE} \setminus \{x, y\}^\triangleright$. For $v \in F_{PAND}$, it holds that $v \in \mathit{Failed}(\pi \cdot xy) \iff v \in \mathit{Failable}(\pi) \wedge \sigma(v) \subseteq \mathit{Failed}(\pi \cdot xy)$. For $v \in F_{SPARE}$, it holds that $v \in \mathit{Failed}(\pi \cdot xy) \iff \mathit{LastClaimed}(\pi, v) \in \mathit{Failed}(\pi \cdot xy) \wedge \mathit{Available}(\pi, v) = \emptyset$.*

With this notion, we see that y can fail immediately with x without changing the semantics.

A note on the connection with the IOIMC semantics Before we conclude the chapter with a small outlook, let us briefly review a major point in our rationale, the compatibility with the IOIMC semantics.

We see three major differences.

1. Immediate vs. delayed failure propagation.



(a) Replacing failure combination by forwarding. (b) Replacing failure forwarding by combination.

Figure 4.12.: Transforming DFTs for compatibility with IOIMC semantics.

2. Early vs. late claiming.
3. Activation semantics (how is activation propagated).

The first difference seems most severe from a practical point of view and is discussed here in greater detail. Early vs. late claiming is a problem which only occurs in nested spares, which is, to the best of our knowledge not used in literature (cf. Section 3.4.13 on page 65) and differences in what exactly is activated are only relevant to non-treelike subtrees - these are again not used in the case studies. We conjecture that in many situations, any differences emerging here could be easily resolved.

We reconsider Figure 3.18a on page 43 which illustrated the effect of delayed failure propagation. We notice that we can replace the DFT by Figure 4.12a - now what was immediate failure propagation according to the earlier defined semantics has become failure forwarding and is therefore delayed. With the help of the notions of preferential basic events and delta-independency, we can argue that in fact, the topmost functional dependencies can be eliminated without affecting the measures of interest (This rewriting is formalised in Chapter 5 on page 113.). Based on this observation, we propose that any DFT which should be analysed according to IOIMC semantics is first transformed as displayed in Figure 4.12b and then rewritten with the framework in our next chapter - based on the semantics presented above. We conjecture that any difference in the outcome of the stochastic analysis is due to the points two and three discussed above. In particular, on all known benchmark instances, calculating the measures-of-interest yields the same result.

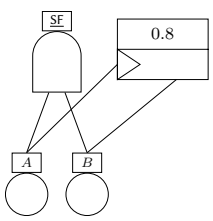
Remark 30. By changing successor relations into functional dependencies, also the claiming and activation mechanisms are affected. We are aware that the method presented is not correct in a formal sense, and is solely discussed here as a starting point for further research.

4.5. Extensions for future work

The presented semantics can be extended to support even more expressive power for DFTs. Due to time- and space restrictions, we only present some brief notes about these extensions.

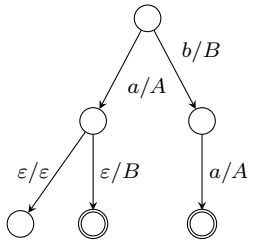
Probabilistic dependencies We recall from Section 3.3.1 on page 34 that probabilistic dependencies are much alike functional dependencies, but only propagate the failure with some discrete probability. To account for this, we can alter the set of dependent events to a multiset from $F_{\text{BE}} \times [0,1]$. To put it differently, instead of a set of dependent sets we have a set of triggered PDEPs. We illustrate the functional transducer and the computation tree for a small example.

Example 4.15. In Figure 4.13a, we show a small DFT consisting of two basic events (A, B). If A fails, then B fails with probability 0.8. In the functional transducer (cf. Figure 4.13b), we see that

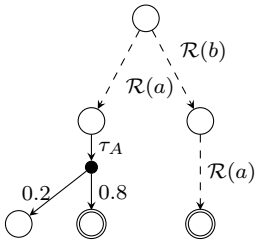


(a) DFT with PDEP

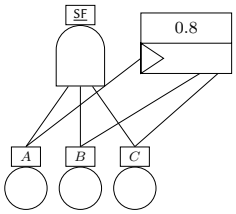
.



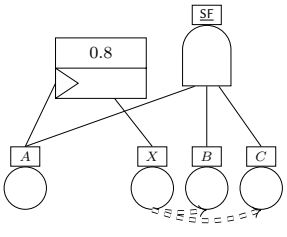
(b) Functional transducer.



(c) Computation tree.



(a) PDEP with multiple dependent events.



(b) Simulating multiple dependent events with binary PDEPs.

Figure 4.14.: PDEPs with multiple dependent events as syntactic sugar

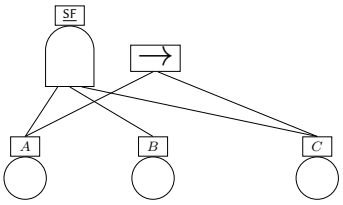
after the failure of A (left path), there are two internal outgoing transitions, one corresponding to B being triggered, and one where we B is not triggered (indicated by the line above B) - notice that the failure of the PDEP is thus non-deterministic. For the latter transition, it is important that this remove B from the set of dependent events. Now the set of dependent events is empty again - and we have only an outgoing transition triggered by the component failure b. The computation tree depicted in Figure 4.13c now combines the transitions ε/B and $\varepsilon/\overline{B}$ into a immediate transitions as before, but instead of a dirac-distribution, use the distribution $\{x \mapsto 0.8, y \mapsto 0.2\}$ where x is the location reached if B is triggered and y if B is not triggered. ▲

It is important to realise that the semantics between a single PDEP with multiple basic events is different frommultiple PDEPs with a a single basic event.

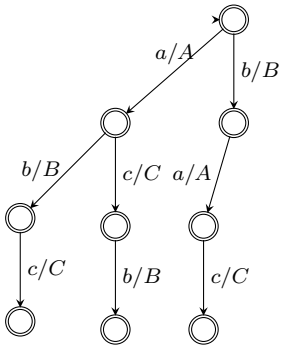
We illlustrate how PDEPs with multiple basic events can be turned into syntactic sugar in Figure 4.14. We see that we introduce a basic event which may be triggered by the PDEP, if it is, all dependent events of the original PDEP are triggered with probability 1 (i.e. by a regular FDEP). If not, non of the original dependent events are triggered.

Sequence enforcers We recall that sequence enforcers require that their successors fail from left to right. Any event chain which breaks this condition is ruled out. We notice that on a functional transducer basis, we can therefore run into a deadlock in which any further event would break the condition. The standard assumes that these deadlocks are allowed, and then the semantics are straightforward to define. Any transition to a location in which a sequence enforcer condition is violated (the sequence enforcer has failed) is removed, i.e., locations with failed sequence enforcers are not reachable. We illustrate this by depicting the functional transducer (Figure 4.15b) for a simple DFT (Figure 4.15a).

Liberal sharing The well-formedness definition given in Definition 4.9 has a bunch of restrictions on spare modules. Their main objective is to prevent multiple spare-gates to claim simultaneously - thereby preventing non-determinism on the level of a single event-chain. These restrictions are in many cases too strong. In particular, they invalidate the "switch" construction in Section 3.4.2 on page 57 (representing barrier failures). We observe that the primary spare module is not independent. However, 1. The primary spare module is activate right from the start, either because it is connected to the top or as it claimed by the active spare. 2. A failure of the module causes only one spare-gate to claim, as no other spares have an immediate cause failure path to the module. In conclusion, the semantics can — and should — be less restrictive. However, this further complicates the proofs.



(a) Simple DFT with sequence enforcer.



(b) Corresponding functional transducer.

Furthermore, the DFTs can be extended by an *Activation Dependency*. Whereas functional dependencies route failures from its trigger to the dependent events, activation throughput activates all dependent events upon the activation of its trigger. With this, we could simplify several DFTs (cf. Figure 3.21 on page 47 and Figure 3.20c on page 47) and eventually ease the rewrite process.

5. Rewriting Dynamic Fault Trees

Fault trees in general, and DFTs in particular, have a tendency of being verbose. First, fault trees are often a form of documentation of the system under consideration. Here, it makes sense to divide systems (trees) in different subsystems (subtrees) based on aspects which might contradict minimality of the DFT, cf. [VS02]. Second, ever more tools deliver computer-generated fault trees based on architecture descriptions, cf. Section 3.4.9 on page 60. The systematic fashion in which these are created often introduce a verbose structure.

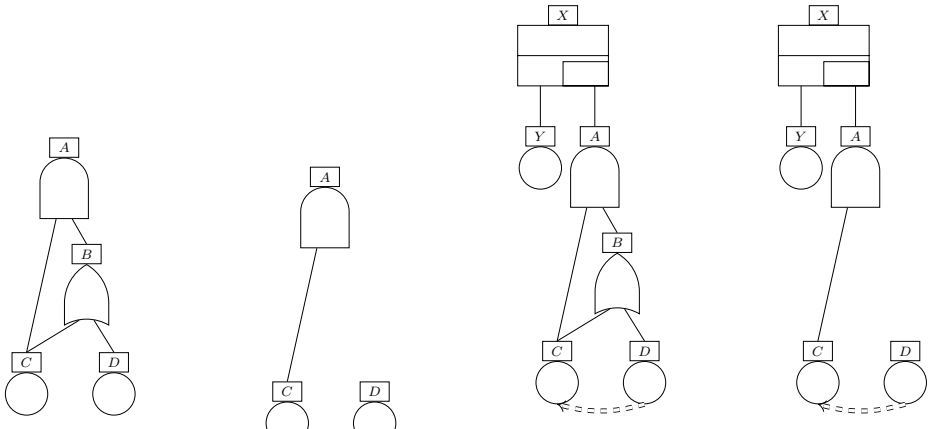
Algorithms for the analysis of DFTs often create an underlying model, e.g. a Markov chain, and then analyse the underlying model. The size of the underlying model is, in general, exponential in the size of the DFT it encodes. Due to the symmetric nature of DFTs observed in practice, state space reduction techniques like (weak) bisimulation are particularly effective, cf. [BCS07c]. However, the size of intermediate model is still potentially very large - in fact - it is the current bottleneck in DFTCalc. Thus, less verbose DFTs encoding the same system potentially improve the performance of DFTCalc.

For the analysis of scenarios with components which have already failed or certainly do not fail, this information might be used to create a less verbose DFT which correctly encodes the system behaviour for the given scenario. The notion of correctness can vary. For some qualitative analysis, it might be important that specific intermediate states are preserved. Here, we are only interested in the quantitative measures we discussed before (cf. Section 4.3.1 on page 100) and therefore, we are not interested in the intermediate states. The goal of this chapter is to formalise rewriting verbose DFTs into smaller DFTs, while maintaining —at least— these measures.

We notice that various rules are not context-free. We give an example that even simple (static) rules are not necessarily context-free.

Example 5.1. Consider Figure 5.1. In Figure 5.1a, we observe that gate A fails if and only if C fails. It seems a general rule that the or-gate (B) does not contribute to the failure behaviour of A. We can therefore remove it as a successor of A, as depicted in Figure 5.1b. In the larger DFT F in Figure 5.1c, we apply the same reasoning, which is perfectly valid. However, by removing the connection, we prevent the activation of D. Thus, in the DFT F' in Figure 5.1d, D is never activated. Assuming D is in cold-standby, D never fails, and thus never triggers C, which also causes F' to have a higher reliability and MTTF than F. ▲

The rule discussed above is the application of the subsumption rule from Boolean algebra. It is extremely effective, as it would allow us to reduce the DFT from 5.1a to a DFT with just element



(a) A small DFT which can be simplified.
(b) The simplified result.
(c) A DFT with the same structure as before
(d) An incorrectly simplified DFT

Figure 5.1.: Rewriting is not context free.

Lattice axioms:

$$a \vee b = b \vee a \qquad a \vee (a \wedge b) = a \qquad (a \vee b) \vee c = a \vee (b \vee c)$$
$$a \wedge b = b \wedge a \qquad a \wedge (a \vee b) = a \qquad (a \wedge b) \wedge c = a \wedge (b \wedge c)$$

Boundedness and distributivity:

$$0 \vee a = a \qquad (a \vee b) \wedge (a \vee c) = a \vee (b \wedge c)$$
$$1 \wedge a = a \qquad (a \wedge b) \vee (a \wedge c) = a \wedge (b \vee c)^{(*)}$$

(*) follows from other axioms.

Table 5.1.: Axiomisation of a bounded distributive lattice $(L, \vee, \wedge, 1, 0)$ with $a, b, c \in L$.

C. However, the example shows us that great care is required when applying the rule. We discuss the context in which such a rule is valid in greater detail during this chapter.

The remainder of this chapter is structured as follows. We first discuss some normal forms, as this guides our expectations for the rewrite rules. We then introduce a general framework for DFT rewriting. We introduce several context restrictions which limit the DFTs on which a rule can be applied. We give the proof obligation for abstract rules to be correct. In the last section, we discuss a selection of rules which we have been embedded in the framework, and proof them correct by showing the aforementioned proof obligations.

5.1. DFTs and normal forms

Let us shortly recall valid rewrite steps for SFTs, which originate from the axioms in a bounded distributive lattice, see Table 5.1. As discussed in Example 5.1, these rules do not hold on general DFTs. However, our rules will allow the same rewriting of static fault trees, as the context of static fault trees allows all the rules to be applied. Instead of explicitly presenting associativity rules, we introduce flattening rules which combine several gates of the same type to one such gate with multiple inputs. Associativity of the gates is easily deduced from that.

Definition 5.1 (Normal forms for static fault trees)**.** Given an SFT $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$. If

- $\mathrm{top} \in$ OR, and
- $\sigma(\mathrm{top}) \subseteq$ AND, and
- $\{\sigma(v) \mid v \in \sigma(\mathrm{top})\} \subseteq$ BE.

then it is in *disjunctive form.* If,

- $\mathrm{top} \in$ AND, and
- $\sigma(\mathrm{top}) \subseteq$ OR, and
- $\{\sigma(v) \mid v \in \sigma(\mathrm{top})\} \subseteq$ BE,

then it is in *conjunctive form.* ■

We recall from Section 4.4 on page 102 that for static fault trees extended with functional dependencies, we can eliminate all functional dependencies by the introduction of or-gates. Thus, the same normal forms as for static fault trees exist.

Proposition 5.1. *For every static fault tree F, possibly extended with FDEPs, equivalent DFTs F_c and F_d exist, with F_c in conjunctive form and F_d in disjunctive form.*

We notice that the DNF for an SFT corresponds to enumerating all minimal cut sets. We could now try to find a normal form which enumerates all minimal cut sets. However, if we consider the DFT encoding a por-gate, we notice that we cannot hope for a DNF with pand-gates instead of and-gates.

Proposition 5.2. *Given the DFT F as depicted in Figure 5.2, for any DFT $F' = (V', \sigma', Tp', \Theta', top')$ with $Dom(Tp) = \{OR, AND, PAND, BE, CONST(\bot), CONST(\top)\}$ s.t. $F \equiv F'$, it holds that*

$$\exists x \in F_{OR}\, \exists y \in F_{PAND}\; y \prec x.$$

We exclude voting-gates, as or-gates are just special voting-gates.

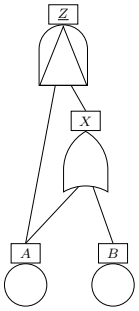


Figure 5.2.: A small DFT encoding the behaviour of a single por-gate.

Proof sketch. We're looking for a DFT F' with the behaviour described below. From Section 4.3.2 on page 100, we know that we can use a trace equivalence here.

π	top $\in$ Failed(π)
ε	✗
A	✓
B	✗
AB	✓
BA	✗

A failure of B before A makes the system infallible. Any equivalent DFT F' thus has a $y \in F'_{\text{PAND}}$, such that $A \prec \sigma(y)_i$ and $B \prec \sigma(y)_j$ with $i < j$. Especially, we can assume that $\sigma(y)_j \in \text{Failed}(B)$. W.l.o.g. we can assume that top $\in$ Failed(π) $\iff$ $y \in$ Failed(π), otherwise, the pand-gate would be dispensable[1], but then there must be another pand-gate, equivalent to y. However, we require that top $\in$ Failed(A), thus $y \in$ Failed(A). Now, consider $\sigma(y)_j$. As $y \in$ Failed(A), we can conclude that $\sigma(y)_j \in$ Failed(A).

We have that $\sigma(y)_j \in$ Failed(A) and $\sigma(y)_j \in$ Failed(B). There thus is a path from $\sigma(y)_j$ to A and a path to B such that all elements on the path fail with A and B, respectively. The pathes have to be merged with an or-gate, as the alternative AND and PAND only fail if the elements on both paths have failed.

We thus have an or-gate x such that $y \prec x$. □

We could easily extend the proposition to include FDEPs and voting-gates. The addition of spare-gates makes a proof significantly harder, but we conjecture that this also holds in the presence of spare-gates.

Ultimately, a structure like in por-gates might yield a normal form, where on the left input we connect a pand-gate which decodes a single sequence, and on the right-hand side, we decode combinations which disable this sequence. The top-level node would still be an and-gate. However, to bring a fault-tree in such a form is not easily done by atomic operations on the graph, instead, it boils down to constructing all sequences, which is as hard as constructing the underlying state space — something which we want to speed up. In the context of this thesis, we therefore abandon the idea of transforming DFTs in a normal form. This is further motivated by the problems which appear with adding functional dependencies and spare gates, which due to timing behaviour and syntactical restrictions do not allow for (relatively) flat normal forms.

As a consequence, we do not have a characterisation of completeness of a rule system.

[1] We omit the formal proof here.

5.2. Rewriting DFTs

As DFTs are just special graphs, a straightforward approach to rewriting DFTs is by graph transformation. We aim to do this analogously to standard graph rewriting, by providing a left-hand (lhs) and a right-hand (rhs) subDFT, matching the lhs in a given host DFT and replacing it by the rhs. To keep rules simple, we want to assure that no other part of the DFT is changed. This motivates us to use DPO graph rewriting.

A DFT and the subDFTs are easily encoded as a labelled graph, by using the underlying graph of the DFT, labelling nodes corresponding to the information encoded in Tp, Θ and top and labelling the edges to encode the ordering. This labelled graph can then be transformed using graph transformation rules. However, standard graph transformation on these graphs does not meet our requirements. In particular, we want to be able to change element types, which is not possible by a straightforward transformation of the subDFTs.

Therefore, we encode the DFT in a more verbose manner, enabling us to use standard graph transformation. An alternative approach would be to define the transformation directly on DFTs, but that would require more new formalisms. Moreover, the reduction to standard graph transformations brings the theory developed here much nearer to the implementation thereof, as discussed in Section 6.1 on page 161.

5.2.1. Graph encoding of DFTs

We present our encoding of a DFT as a labelled graph. The idea is that instead of labelling the elements (nodes in the graph) directly, we attach secondary nodes for each DFT element, which we call the *carry nodes*. These nodes are then used to "carry" the information of the element. With this construction, we can match any subDFT in a DFT purely based on the graph structure, while maintaining the element type in the graph. Later, for the rewriting, if we want to restrict an element to a given type, we add the carry node to the left-hand side of the rule.

Remark 31. An alternative approach would be the use of self-loops as carries. However, without the introduction of multi-edges, using self-loops is not suitable for handling multiple different labels, and matching subsets of such labels.

The following definition is used to construct edge labels to encode the ordering.

Definition 5.2. Let $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ be a DFT. A mapping $o : E(\sigma) \to \mathbb{N}$ is *edge order preserving* if for each $v \in V_A$ with $1 \leq i < j \leq \deg(v)$ it holds that

$$o((v, \sigma(v)_i)) < o((v, \sigma(v)_j)).$$

If for all $v \in V_A$ and $1 \leq i \leq \deg(v)$ we have $o(v, \sigma(v)_i) = i$, then we call o trivial. ■

We first give the definition and continue then with an example illustrating the encoding.

Definition 5.3. Labelled graph of a DFT Given a DFT $F_\Omega = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$. Let o be the trivial edge order preserving mapping. A labelled graph $\mathcal{F} = (V_\mathcal{F}, E_\mathcal{F}, l_\mathcal{F})$ over $((\Omega \cup \{d\} \cup \mathsf{Gates} \cup \mathsf{Leafs} \cup \{0, \mathrm{top}\}), \mathbb{N})$ is the *DFT graph representation for* F with

- $V_\mathcal{F} = (V \times \{1, 2\}) \cup \{(v, 3) \mid v \in F_{\mathrm{BE}}\} \cup \{v_{\mathrm{top}}\}$,
- $E_\mathcal{F} = \{((v, 1), (v', 1)) \mid (v, v') \in E(\sigma)\}$
  $\cup \{((v, 1), (v, 2)) \mid v \in V\}$
  $\cup \{((v, 1), (v, 3) \mid v \in F_{\mathrm{BE}}\}$
  $\cup \{(\mathrm{top}, 1), (v_{\mathrm{top}}, 2)\}$

- $\mathrm{lv}_{\mathcal{F}} = (v,1) \mapsto 0$ ■
  $\cup\, (v,2) \mapsto \mathrm{Tp}(v)$
  $\cup\, (v,3) \mapsto \begin{cases} \Theta^{-1}(v) & \text{if defined for } v \\ d & \text{else.} \end{cases}$
  $\cup v_{\text{top}} \quad \mapsto \text{top}$

  $\mathrm{le}_{\mathcal{F}} = ((v,1),(v',1)) \mapsto o((v,v'))$
  $\cup\, ((v,1),(v,2)) \mapsto 0$
  $\cup\, ((\text{top},1), v_{\text{top}}) \mapsto 0$

Example 5.2. In Figure 5.3 we depict a DFT and its encoding as labelled graph. The encoding has three element nodes, with on their left side their carry nodes. Moreover, the topmost node has a carry node which identifies it as the top node, while the two basic elements have extra carry nodes for the attachment function. All carry nodes are connected via edges labelled with 0, while the other two edges have non-zero labels encoding the ordering. ▲

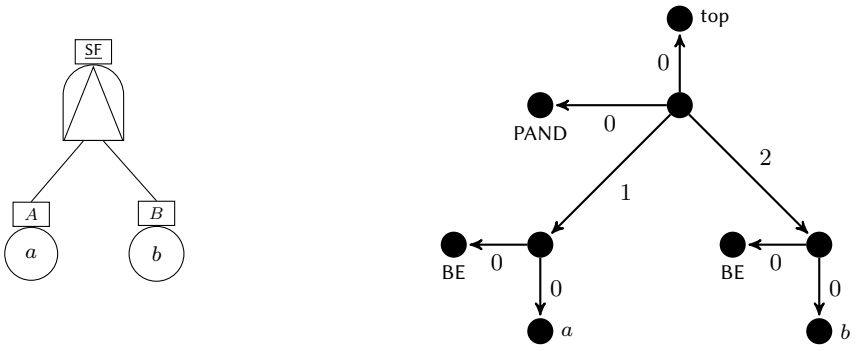


Figure 5.3.: A DFT and the labelled graph representation of a DFT

In order to define transformations on DFTs via their encoded labelled graph representation, we also need the decoding.

We first partition the nodes of any labelled graph $\mathcal{F}$ over $((\Omega \cup \mathsf{Gates} \cup \mathsf{Leafs} \cup \{0, \text{top}\}), \mathbb{N})$. The set Carries of carry nodes is defined as

$$\mathrm{Carries}_{\mathcal{F}} = \{v \in V_{\mathcal{F}} \mid \exists (v', v) \in E_{\mathcal{F}}.\ \mathrm{le}_{\mathcal{F}}(v', v) = 0\}.$$

Moreover, the set $\mathrm{Elements}_{\mathcal{F}}$ of element nodes is defined as $\mathrm{Elements}_{\mathcal{F}} = V_{\mathcal{F}} \setminus \mathrm{Carries}_{\mathcal{F}}$.

Definition 5.4 (DFT of a labelled graph representation)**.** Given a labelled graph $\mathcal{F} = (V_{\mathcal{F}}, E_{\mathcal{F}}, l_{\mathcal{F}})$ over $((\Omega \cup \mathsf{Gates} \cup \mathsf{Leafs} \cup \{0, \text{top}\}), \mathbb{N})$. If the following conditions all hold, we call $\mathcal{F}$ a valid DFT encoding.

- Carry nodes do not have outgoing edges and only one incoming edge.

  $$\forall v \in \mathrm{Carries}_{\mathcal{F}}.\ \mathrm{Out}(v) = \emptyset \wedge |\mathrm{In}(v)| = 1.$$

- Each element node has at least a carry node for the type connected.

  $$\forall v \in \mathrm{Elements}_{\mathcal{F}}.\ \exists v' \in \mathrm{OutV}(v) \cap \mathrm{Carries}.\ \mathrm{lv}_{\mathcal{F}}(v') \in \mathsf{Gates} \cup \mathsf{Leafs}.$$

- There exist as many carry nodes for the type as there exist elements.

  $$|\mathrm{Elements}_{\mathcal{F}}| = |\{v \in \mathrm{Carries}_{\mathcal{F}} \mid \mathrm{lv}_{\mathcal{F}}(v) \in \mathsf{Gates} \cup \mathsf{Leafs}\}|.$$

- Exactly the elements with a type carry labelled as a basic element also have a carry representing the attachment.

$$\begin{aligned}&\forall v \in \text{Elements}_{\mathcal{F}}.\\ &\quad \exists v' \in \text{OutV}(v) \cap \text{Carries}.\ \text{lv}_{\mathcal{F}}(v') \in \text{BE} \iff\\ &\quad \exists v'' \in \text{OutV}(v) \cap \text{Carries}.\ \text{lv}_{\mathcal{F}}(v'') \in \Omega \cup \{d\}\end{aligned}$$

- Carries are not labelled 0.

$$\forall v \in \text{Elements}_{\mathcal{F}}.\ \text{lv}_{\mathcal{F}}(v) \neq 0.$$

- Element nodes are labelled with 0.

$$\forall v \in \text{Elements}_{\mathcal{F}}.\ \text{lv}_{\mathcal{F}}(v) = 0.$$

- Each component failure is used as a label as most once, i.e. for all $\omega \in \Omega$,

$$\nexists \{v, v'\} \subseteq V_{\mathcal{F}}.\ \text{lv}_{\mathcal{F}}(v) = \omega = \text{lv}_{\mathcal{F}}(v')$$

- There is exactly one node labelled as top node.

$$\exists v \in V_{\mathcal{F}}.\ \text{lv}_{\mathcal{F}}(v) = \text{top} \wedge \forall v' \in V_{\mathcal{F}} \setminus \{v\}.\ \text{lv}_{\mathcal{F}}(v') \neq \text{top}$$

- Two edges with the same source are labelled differently or 0, i.e., for all $\{(s,t),(s,t')\} \subseteq E_{\mathcal{F}}$,

$$\text{le}_{\mathcal{F}}(s,t) = \text{le}_{\mathcal{F}}(s,t') \implies \text{le}_{\mathcal{F}}(s,t) = 0.$$

We define F the decoding of $\mathcal{F}$ with $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$ over $\Omega' = \{\omega \in \Omega \mid \exists v \in V_{\mathcal{F}}.\ \text{lv}_{\mathcal{F}}(v) = \omega\}$, s.t.

- $V = \text{Elements}_{\mathcal{F}}$.
- σ s.t. for all $v \in V$, $\sigma(v) = v_1 \dots v_m$ with

$$\begin{aligned}&\forall 1 \leq i \leq m.\ (v, v_i) \in E_{\mathcal{F}}, \text{ and}\\ &\forall 1 \leq i \leq m.\ \text{le}_{\mathcal{F}}(v, v_i) \neq 0, \text{ and}\\ &\forall 1 \leq i < j \leq m.\ \text{le}_{\mathcal{F}}(v, v_i) < \text{le}_{\mathcal{F}}(v, v_j)\end{aligned}$$

- $\text{Tp}(v) = x \exists v' \in \text{OutV}(v) \cap \text{Carries}.\ \wedge x = \text{lv}_{\mathcal{F}}(v') \in \mathsf{Gates} \cup \mathsf{Leafs}$
- $\Theta(\omega) = v$ s.t. $\{v\} = \text{InV}(v')$ where $\text{lv}_{\mathcal{F}}(v') = \omega$.
- $\text{top} = v$ s.t. $\{v\} = \text{InV}(v')$ where where $\text{lv}_{\mathcal{F}}(v') = \text{top}$. ■

For each DFT, subsequent application of encoding and decoding keeps the DFT intact.

Corollary 5.3. *Given a DFT F and a labelled graph representation $\mathcal{F}$ for F. Let F' be the DFT encoded by $\mathcal{F}$. It holds that $F = F'$.*

The proof involves a straightforward check of all conditions and applying the definitions of both directions. We do not present the full proof here.

5.2.2. Defining rewriting on DFTs

The idea we formalise here is captures in Figure 5.4. Given a DFT we want to rewrite with a given rule, we first encode the DFT as a labelled graph as discussed above. As we want to define the rewrite rule on DFTs, we also need to encode the rule as a rewrite rule for labelled graphs. After the application of the rewrite rule, we decode the labelled graph back to a DFT, also discussed above.

The core of the rewrite rule consists of the two subDFTs, representing the left- and right hand side of the rule.

Definition 5.5. Given a DFT $F = (V, \sigma, \text{Tp}, \Theta, \text{top})$. A *subDFT of F* is a tuple $X = (V_X, \sigma_X, \text{Tp}^\sharp_X)$ with $V_X \subseteq V$, $\sigma_X = \sigma_{|V_X}$ and $\text{Tp}^\sharp_X = \text{Tp}_{|Z}$ where $Z \subseteq V_X$. Moreover, we call any $Y =$

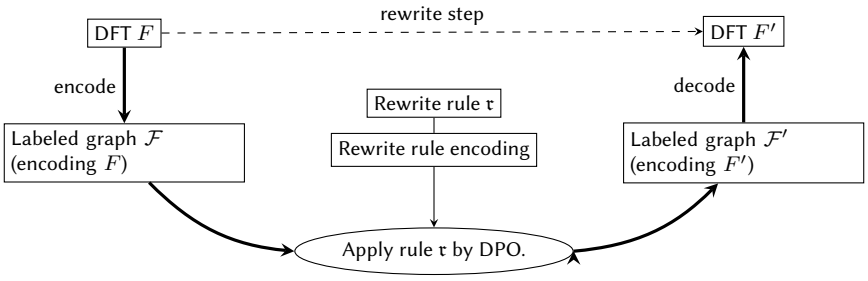


Figure 5.4.: Formalise DFT rewriting via standard graph rewriting

$(V_Y, \sigma_Y, \mathrm{Tp}_Y^\sharp)$ a *subDFT* if there exists a DFT F' such that Y is a subDFT of F'. We call Y *well-formed* if there exists a well-formed DFT F' such that Y is a subDFT of F'. ■

We use notions and notations for DFTs also on subDFTs.

Encoding a subDFT as a labelled graph is defined analogously to DFTs. For elements where the type mapping is undefined, we simply do not add type carries. We illustrate this after the introduction of rewrite rules.

A rewrite rule on DFTs consists, like DPO rewrite rules, of an interface, left- and right hand side graphs, a set of homomorphisms. Additionally, rewrite rules on DFTs contain a context restriction, which defines a set of DFTs on which the rule may be applied. We split the interface into an input (sinks in the graph) and an output interface. Intuitively, only nodes from the input interface may have successors which are not matched, and input elements do not have successors in the rewrite rule. Types of the input elements may not change. Nodes from the input and the output interface may have unmatched predecessors. Output interface nodes should not be FDEPs, as FDEPs cannot have predecessors. Contrary to DPO, the interface of a rewrite rule for DFTs does not contain edges. The additional homomorphisms are due to the fact we distinguish between input and output interface.

Definition 5.6 (Rewrite rule)**.** A *rewrite rule on DFTs* is a tuple $(V_\mathfrak{i}, V_\mathfrak{o}, L, R, H, \mathfrak{C})$ consisting of

- A set of *input elements* $V_\mathfrak{i}$
- A set of *output elements* $V_\mathfrak{o}$
- A *matching subdft* $L = (V_L, \sigma_L, \mathrm{Tp}_L^\sharp)$
- A *result subdft* $R = (V_R, \sigma_R, \mathrm{Tp}_R^\sharp)$
- Four graph morphisms $H = \{h_i, h_o, h_l, h_r\}$, where V_K denotes $V_\mathfrak{i} \cup V_\mathfrak{o}$.
  - Two embeddings $h_i \colon V_\mathfrak{i} \to V_K$ and $h_o : V_\mathfrak{o} \to V_K$.
  - One embedding $h_l \colon V_K \to L$
  - An homomorphism $h_r \colon V_K \to R$ such that $h_r|_{V_\mathfrak{i}}$ is injective.
- A context restriction set $\mathfrak{C} = \{(\mathfrak{C}_i, \zeta_i) \mid i \in \mathbb{N}\ \mathfrak{C}_i\mathrm{DFT}, \zeta_i \colon L \to V\}$ of *embargo DFTs* where ζ_i is a homomorphism for each $i \in \mathbb{N}$.

such that the following conditions hold.

- The input and output elements are disjoint.

$$h_i(V_\mathfrak{i}) \cap h_o(V_\mathfrak{o}) = \emptyset$$

- The left-hand side elements without children are exactly the input elements.

$$h_{\mathrm{Lab}}(V_\mathfrak{i}) = \{v \in L | \sigma(v) = \emptyset\}.$$

- No input element is mapped to a right-hand side which has successors.

$$\forall v \in h_r(V_\mathfrak{i}).\ \sigma(v) = \emptyset$$

- At least all the elements which are not input elements are typed.

$$V_L \setminus h_{\mathrm{Lab}}(V_\mathfrak{i}) \subseteq \mathrm{Dom}(\mathrm{Tp}_L^\sharp) \wedge V_R \setminus h_r(V_\mathfrak{i}) \subseteq \mathrm{Dom}(\mathrm{Tp}_R^\sharp).$$

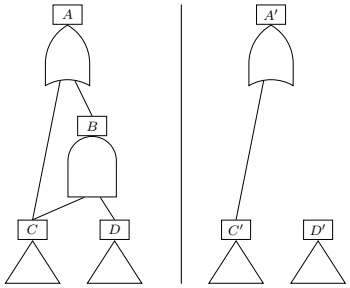


Figure 5.5.: The L and R subDFTs of the subsumption rule.

- Typing of input elements in the matching and the result graph agree.

$$\forall v \in V_{\mathfrak{i}}.\ \mathrm{Tp}^{\sharp}_{\mathrm{Lab}}(h_{\mathrm{Lab}}(v)) = \mathrm{Tp}^{\sharp}_{R}(h_r(v)).$$

- Output elements are not typed as FDEPs.

$$\forall v \in V_{\mathfrak{o}}.\ \mathrm{Tp}^{\sharp}_{\mathrm{Lab}}(h_{\mathrm{Lab}}(v)) \neq \mathrm{FDEP} \neq \mathrm{Tp}^{\sharp}_{R}(h_r(v))$$ ■

We write $(L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \mathfrak{C})$. We abuse the notation and use h_l (h_r) also to denote the embedding from V_K into $\mathcal{L}$ ($\mathcal{R}$).

Definition 5.7 (DPO variant of a rewrite rule for DFTs)**.** Given a rewrite rule on DFTs $\mathfrak{r} = (V_{\mathfrak{i}}, V_{\mathfrak{o}}, L, R, \{h_i, h_o, h_l, h_r\}, \mathfrak{C})$, we define the graph rewrite rule $(\mathcal{L}, V_{\mathfrak{i}} \cup V_{\mathfrak{o}}, \mathcal{R}, h_l, h_r\}$ as the corresponding rewrite rule. ■

The definition above does not add the carries of the interface elements to the interface of the graph rewrite rule. This is not of importance, as the carries are only connected to these interface nodes. With this graph rewrite rule, the carries are deleted and added afterwards, thereby not changing them. Notice that the attachment function is untouched. Thus, we cannot delete any basic events with an attached component failure.

Furthermore, the definition above does not take the context restriction into account. We do not treat the enforcement of the context restriction here. As the host graph is finite, there exist only finitely many graphs in the context restrictions which potentially describe the host graph. For the formal treatment here, we assume that the context restrictions on the host graph are enforced by an additional check. For the application in Chapter 6 on page 161, the used tool set Groove supports the context restrictions as formulated here.

Example 5.3. In this example, we cover the subsumption rule discussed already in Example 5.1, where we argued that the rule is not context-free.

First, let us capture the left and right subDFTs which are at the heart of the rule. We display them in Figure 5.5. A triangle depicts an untyped element.

We define $V_{\mathfrak{i}} = C, D$ and $V_{\mathfrak{o}} = A$. The homomorphisms h_i, h_o, h_l are trivial embeddings (the identifiers share the same name). The homomorphism h_r is given by $A \mapsto A', C \mapsto C', D \mapsto D'$. Furthermore,

$$L = (\{A, B, C, D\}, \{A \mapsto CB, B \mapsto CD, C \mapsto \varepsilon, D \mapsto \varepsilon\}, \{A \mapsto \mathrm{AND}, B \mapsto \mathrm{OR}\})$$

and

$$R = (\{A', C', D'\}, \{A' \mapsto C', C' \mapsto \varepsilon, D' \mapsto \varepsilon\}, \{A' \mapsto \mathrm{AND}\}).$$

The context restriction is given as

$$\mathfrak{C} = \{F \text{ DFT} \mid \exists x \in \sigma^*(\zeta(D))\theta^*(x) \subseteq \sigma^*(\zeta(D)) \cup \{\zeta(D)\} \vee \theta(\zeta(D)) \neq \emptyset\}$$

All conditions from Definition 5.6 are fulfilled.

We notice that we cannot match B on an element with other successors than the element matched by A. If it would be allowed, we would gain a dangling edge, as we remove the ele-

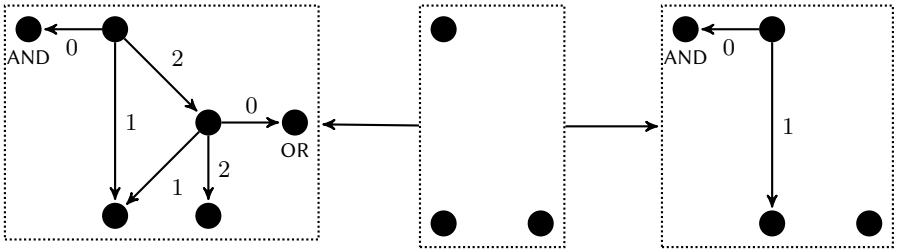


Figure 5.6.: Graph representation for subsumption rule.

ment matched by B. For the context restriction, we assume a sufficient strong criterion to ensure that we do not run into the same problems as in Example 5.1 on page 113. For this example, we use a criterion much stronger than required. The criterion used here guarantees that the element matched by D does not have any predecessors after rewriting and all successors of this element have successors only in D. Therefore, either the top-level element is in the subtree, but then the subtree (a module) is represented by the top level element and is always active. Otherwise, it does not matter whether the subtree is ever activated or not, as it is futile, exactly as the complete subtree underneath it.

The corresponding graph rewrite rule is given in Figure 5.6. The actual homomorphisms between the nodes is given by their relative position in the graph. We see that the two elements without type do not have carries attached. The graph rewriting removes everything but the tree nodes corresponding to the elements. In particular, it also removes the type of the topmost element, which might not be expected. On the right-hand side, this type is added again.

▲

Given a rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$, if the element hierarchy of L restricted to $h_{\text{Lab}}(V_\mathfrak{i} \cup V_\mathfrak{o})$ is a superset of the element hierarchy of R restricted to $h_r(V_\mathfrak{i} \cup V_\mathfrak{o})$, then $\mathfrak{r}$ is *hierarchy conservative*. If the module relation of L restricted to $h_{\text{Lab}}(V_\mathfrak{i} \cup V_\mathfrak{o})$ is a superset of the module relation of R restricted to $h_r(V_\mathfrak{i} \cup V_\mathfrak{o})$, then $\mathfrak{r}$ is *module conservative*.

Definition 5.8 (Match)**.** Given a DFT F with its graph representation $\mathcal{F}$ and rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ and a injective graph morphism κ from $\mathcal{L}$ into $\mathcal{F}$ such that

1. The dangling edge condition holds.
2. All successors of an output element matched vertex are also in the rule, i.e.

$$\forall v \in V_\mathfrak{o}\ |\text{OutV}(v)| = |\text{OutV}(\kappa(v))|$$

then $\kappa(\mathcal{L})$ is a *matched graph* and $\kappa(V_\mathfrak{i} \cup V_\mathfrak{o})$ is the *match glue*. We call κ the *rule match morphism* and call $\mathfrak{r}$ a matching rule. ■

In the definition for a match, we ignored the context restrictions. This is captured by the definition of a successful match.

Definition 5.9 (Effective embargo)**.** Given a matching rule with rule match morphism κ. Then an embargo (graph) $\mathfrak{C}(i)$ is *effective* if there exists an injective homomorphism h from $\mathfrak{C}(i)$ to G with

$$h(\zeta_i(V_\mathfrak{i})) = \kappa(h_i(V_\mathfrak{i})) \text{ and } h(\zeta_i(V_\mathfrak{o})) = \kappa(h_o(V_\mathfrak{o}))$$ ■

Definition 5.10 (Successful Match)**.** A matching rule is successful, if no i exists such that $\mathfrak{C}_i$ is an effective embargo. ■

Definition 5.11 (Rewrite step)**.** Let $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule and F and F' DFTs with graph representations $\mathcal{F}$ and $\mathcal{F}'$. Let $\mathfrak{r}$ be successfully matched in F by the homomorphism κ. Then F is rewritten by $\mathfrak{r}$ using κ to F' if $V_K\mathcal{L}D\mathcal{F}/V_K\mathcal{R}D\mathcal{F}'$ is a DPO with $D = (V_\mathcal{F} \setminus (\kappa(V_\mathcal{L}) \setminus \kappa(V_K)), E_\mathcal{F} \setminus \kappa(E_\mathcal{L}))$.

The tuple (F, r, κ, F') is a a *rewrite step*. The homomorphism $\mathcal{R} \rightarrow \mathcal{F}'$ is denoted by η. ■

It is not yet clear that the outcome of this procedure again encodes a DFT.

Theorem 5.4. *Using the notation from Definition 5.11, $\mathcal{F}'$ is a valid DFT encoding.*

Before we proof this, we introduce some auxiliary notation.

Given a rewrite step $(F, (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C}), \kappa, F')$ with a DPO $V_K\mathcal{L}D\mathcal{F}/V_K\mathcal{R}D\mathcal{F}'$, we write $\nu : D \nrightarrow F$ and $\nu' : D \nrightarrow F'$ to represent the mappings from the common graph to the input and the output DFT, respectively. We abuse the notation and use η also to denote the mapping from R to F', induced by the homomorphism $\eta\colon \mathcal{R} \to \mathcal{F}'$.

In the following, we use the term *original element* to refer to elements in F' which already occurred in F. This is formalised as those elements of F' who, in the graph representation, have a preimage in D w.r.t. the homomorphism from D to $\mathcal{F}'$, i.e.

$$\text{original elements } = \{v \in F' \mid \nu'^{-1}(v) \neq \bot\}$$

We use *new (old) element* to refer to elements which do not have a preimage in D (R), i.e.

$$\text{new elements} = \{v \in F' \mid \nu'^{-1}(v) = \bot\}$$
$$\text{old elements} = \{v \in F' \mid \eta^{-1}(v) = \bot\}$$

Elements which have a preimage in D and in R are called *glued elements*. Please, notice that each original element is either an old or a glued element.

We notice that for any element in $v' \in \nu'(D \setminus \kappa(V_\mathfrak{i} \cup V_\mathfrak{o})))$ that for $v = \nu'^{-1} \circ \nu(v')$, $\sigma_F(v) = \nu'^{-1} \circ \nu(\sigma_{F'}(v'))$, $\mathrm{Tp}(v) = \mathrm{Tp}(v')$, $\Theta(v) = \Theta(v')$. Therefore, we allow any $v \in \nu'(D \setminus (V_\mathfrak{i} \cup V_\mathfrak{o}))$ to be denoted with either v or $\nu'^{-1} \circ \nu(v)$, as well as the other way around.

The application of the rule changes the labelled graph. However, all properties which ensure that the labelled graph encodes a DFT are preserved.

Proof sketch for Theorem 5.4. We argue for each of the conditions from Definition 5.4.

- *Carry nodes do not have outgoing edges and only one incoming edge.*
  Edges are only removed inside $\kappa(\mathcal{L})$. So no connections to carry nodes are removed unless the corresponding nodes are also removed. Other carry nodes are not connected to any elements. No outgoing edges to carry nodes are introduced as $\mathcal{F}$ and $\mathcal{R}$ are valid encodings themselves.
- *Each element node has at least a carry node for the type connected.*
  Old nodes have this as $\mathcal{F}$ is a valid DFT encoding. New nodes have a type added as $\mathcal{R}$ is a valid subDFT encoding. Nodes with a preimage in $\kappa(V_\mathfrak{i})$ either have a type described (and thus added) in R, or the type is not specified, but then it was not removed either in L and is thus in D.
- *There exist as many carry nodes for the type as there exist elements.*
  Together with the two points before, it suffices to show that each carry node is connected with at most one type-carry node. New or old nodes have this as $\mathcal{F}$ is a valid DFT encoding. Nodes with a preimage in $\kappa(V_\mathfrak{o})$ have no type, as it is removed with L, which has types for all elements in $V_\mathfrak{o}$. Nodes with a preimage in $\kappa(V_\mathfrak{i})$ either have a type described (and thus removed) in L, or the type is not specified, but then it not added either in R.
- *Exactly the elements with a type carry labelled as a basic element also have a carry representing the attachment.*
  For new and for old nodes, this follows from the valid encoding of $\mathcal{R}$ and $\mathcal{F}$. For elements with a preimage in $\kappa(V_\mathfrak{i})$, we never change the type, so exactly the nodes which encoded a basic element still encode a basic element and we did not touch the attachment carries. For elements with a preimage in $\kappa(V_\mathfrak{o})$, we have that they are not basic elements before, otherwise they would be in $V_\mathfrak{i}$ as well, which is a contradiction. Thus, we do not have attachment carries on $V_\mathfrak{o}$ which are no basic elements. We only add attachment carries to those elements which are new basic elements.
- *Carries are not labelled* 0, and
- *element nodes are labelled with* 0.
  Follow directly from the fact that all nodes in $\mathcal{F}'$ originate from a valid DFT encoding.
- *Each component failure is used as a label as most once.*
  Follows directly from the fact that we do not add any attachments, as the component failures do not even occur in the rewrite rule.
- *There is exactly one node labelled as top node.*

The top level element is in D, the carry node using this is not touched. The encoding of subDFTs does not have carry nodes for the the top.

- *Two edges with the same source are labelled differently or* 0.
  For new and for old nodes, this follows from the valid encoding of $\mathcal{R}$ and $\mathcal{F}$. For nodes with a preimage in $\kappa(V_\text{i})$, notice that the nodes in $h_r(V_\text{i})$ do not have any successors other than their carries (labelled 0) and in D, the property above holds. For nodes with a preimage in $\kappa(V_\text{o})$, notice that these nodes do not have any successors in D, and that in $\mathcal{R}$ the property above holds.

□

5.2.3. Preserving syntax

Applying well-formed rewrite rules on well-formed DFTs should result in well-formed DFTs. We first give two counterexamples which show that this property is not trivially fulfilled. The first is straightforward and uncovers problems which are caused by the syntactic restrictions for spare components.

Example 5.4. Consider the DFT from Figure 5.7a. The DFT is obviously constantly failed. The FDEP is the only FDEP, so it is certainly not in conflict with other FDEPs, moreover, the trigger and the dependent event are not below a dynamic gate. We therefore want to eliminate the FDEP, yielding the DFT depicted in Figure 5.7b.

The propagation of the constant fault via in the given context is trivially generalised to a rule[1]. Consider Figure 5.7c and Figure 5.7d, depicting L and R, respectively. Notice that L and R are well-formed subDFTs.

We apply this rule, which is later captured by Rewrite rule 24, to the graph in Figure 5.7e. The result depicted in Figure 5.7f is not a well-formed DFT, as first, constant faults are not allowed as primary spare components, and second, the constant fault is connected directly to the top-level and to the spare. ▲

The second rule might come as a surprise, as one would not expect cyclic DFTs to be constructed. That this is possible is a straight consequence of the fact that acyclicity is a global criterion, while the acyclicity of the host DFT and the rule graphs are just local criteria.

Example 5.5. Consider the DFT from Figure 5.8a on page 125. Based on the same arguments as in Example 5.4 above, we want to rewrite the FDEP. As a result, the DFT can be rewritten to the DFT of Figure 5.8b.

We want to use this as a general rule. We use the subDFTs from Figure 5.8c and Figure 5.8d as L and R, respectively. Again, L and R are well-formed.

Now, we apply the rule on the DFT in Figure 5.8e. The result is depicted in Figure 5.8f. We introduced a cycle. ▲

We discuss the actually used form of this rule as Rewrite rule 24.

We conclude that there exist rules with practical relevance that are only applicable on a restricted class of DFTs. We propose two different solutions.

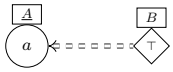
1. An *a-posteriori check*, which means that after each rewrite step, we check whether the result is a well-formed DFT. If it is not, the rule is revoked and we go back to the original host DFT.
2. An *a-priori check*, which means that before we apply the rule, we check whether the result will be a well-formed DFT, based on the host DFT and the rewrite rule to be applied.

As the framework is used as basis for a rewriting algorithm, we prefer the a-priori conditions which prevent the syntax check. In general, this is very hard. We therefore restrict the rules for which we define this a-priori conditions. For all others, we require the a-posteriori checks. The next theorem shows that we can leave out the a-posteriori check for a large set of rules. It also shows how tedious it may be to define such a check for an even larger class of DFTs and rules.

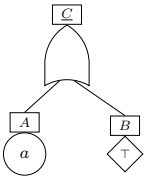
Theorem 5.5. *Given a rewrite rule* $\mathfrak{r} = (L \leftarrow (V_\text{i} \cup V_\text{o}) \rightarrow R, \mathfrak{C})$ *and a DFT* F *with a successfully matched graph* L *by* κ.

We assume that $R_{SPARE} = \emptyset$ *and* $|\{[v]_{\bowtie_L} \mid v \in V_L\}| \leq 2$. *If all of the following conditions hold, then* F *rewritten by* $\mathfrak{r}$ *using* κ *is a well-defined DFT.*

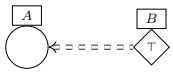
[1] Indeed, the only difference is that in the rule, A is not required to be the top level element



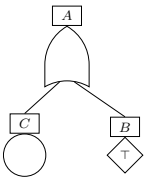
(a) Initial DFT which can be simplified according to specific rule.



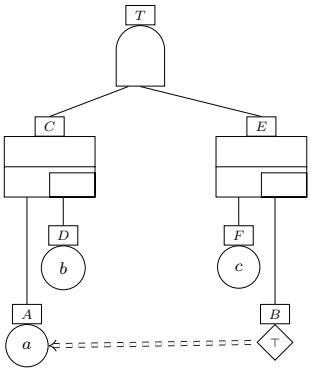
(b) The resulting DFT after applying the rule.



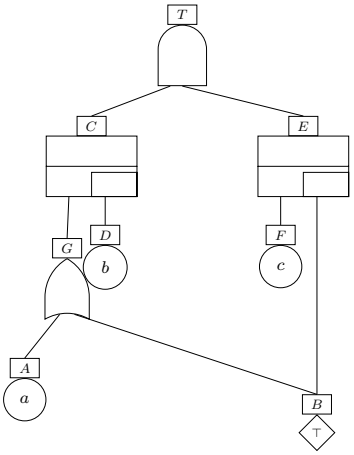
(c) The matching subDFT.



(d) The result subDFT.



(e) A DFT with a similar construct



(f) An invalid result after applying the rule.

Figure 5.7.: Invalid syntax due to merging spare components.

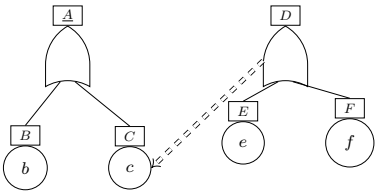
1. *Either*
   - $\mathfrak{r}$ *is hierarchy conservative, or*
   - *considering* $S = \{(x, y) \in V_\mathfrak{o} \times V_\mathfrak{i} \mid x \prec_R y \wedge x \not\prec_L y\}$*, we have that either*
     - $\forall\{(x, y), (x', y')\} \subseteq S$*,* $y = y'$*, or*
     - *there exists an* $w \in V_R$ *with* $w \prec h_r(y)$ *for all* $(x, y) \in S$ *and* $h_r(x) \prec w$*,*

   and the following holds:
     - $\{(\kappa(y), \kappa(x)) \mid (x, y) \in S\} \cap \prec_F = \emptyset$*.*
2. *Either*
   a) $\mathfrak{r}$ *is module conservative, or*
   b) *Merging modules leads to at most one module, formally*

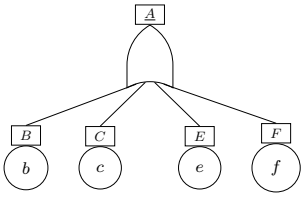
$$\begin{aligned}
&\exists v, v' \in V_\mathfrak{i} \cup V_\mathfrak{o} \text{ s.t.} \\
&\quad [v]_{\bowtie_L} \neq [v']_{\bowtie_L} \wedge [h_r(v)]_{\bowtie_R} = [h_r(v')]_{\bowtie_R} \wedge \ \forall w, w' \in V_\mathfrak{i} \cup V_\mathfrak{o} \\
&\qquad ([w]_{\bowtie_L} \neq [w']_{\bowtie_L} \wedge \{[w]_{\bowtie_L}, [w']_{\bowtie_L}\} \neq \{[v]_{\bowtie_L}, [v']_{\bowtie_L}\}) \\
&\qquad\quad \implies \\
&\qquad ([h_r(w)]_{\bowtie_R} \neq [h_r(w')]_{\bowtie_R} \vee [h_r(w)]_{\bowtie_R} = [h_r(v)]_{\bowtie_R}),
\end{aligned}$$

   and either of the following holds:
   i. *In the host DFT, it already was a single module:*

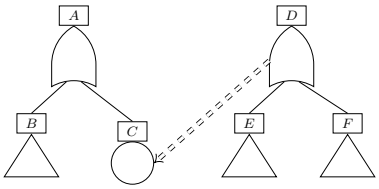
$$\forall v, v' \in V_\mathfrak{o} \cup V_\mathfrak{i}.\ x \not\bowtie_L x' \wedge x \bowtie_R x' \implies \kappa(x) \bowtie_F \kappa(x')$$



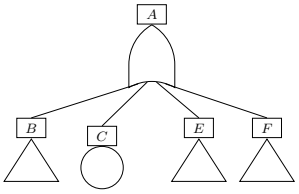
(a) A DFT which can be simplified with a specific rule.



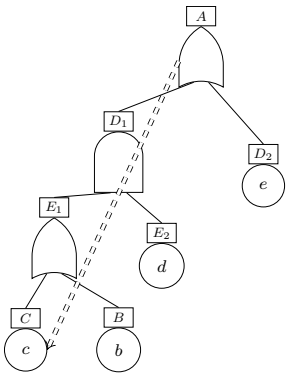
(b) The result after simplification.



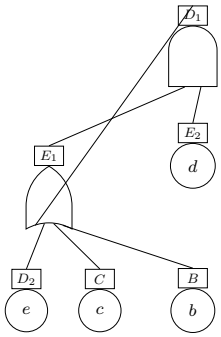
(c) The matching subDFT.



(d) The result subDFT.



(e) Another DFT on which the rule is applicable.



(f) The result has a cycle and thus it is not a well-formed DFT

Figure 5.8.: Introduction of cycles by rewrite rules.

ii. At most one of the old components were connected to top or to a spare:

$$\exists x \in V_F \text{ s.t. } x = top_F \vee x \in \sigma(F_{SPARE}) \wedge \exists y \in [v]_{\bowtie_L}\ x \bowtie_F \kappa(y)$$
$$\implies$$
$$\forall y' \in V_{\mathfrak{o}} \cup V_{\mathfrak{i}}\ y' \not\bowtie_L v \wedge h_r(y') \bowtie_R h_r(v)$$
$$\nexists x' \in V_F \text{ s.t. } x' = top_F \vee x' \in \sigma(F_{SPARE}).\ x' \bowtie_F \kappa(y')$$

. Moreover, the other component does not include a given-failure element, formally (by adding this to the right-hand side of the formula above).

$$\wedge\ \nexists c \in F_{CONST(\top)} c \bowtie_F y'$$

iii. The representants are merged as well: $h_r(x) = h_r(x')$ *with* $x, x' \in V_{\mathfrak{o}}$ *and* $v \bowtie_L x$ *and* $v' \bowtie_L x'$ *and* $\theta(\kappa(x)) \cap SPARE_F \neq \emptyset \neq \theta(\kappa(x')) \cap SPARE_F$.

3. $\{v \in V_{\mathfrak{i}} \cup V_{\mathfrak{o}} \mid \exists v'.\ Tp^{\sharp}_R(v') = CONST(\top) \wedge h_r(v) \bowtie_R v'\}$
$\subseteq \{v \in V_{\mathfrak{i}} \cup V_{\mathfrak{o}} \mid \exists v'.\ Tp^{\sharp}_L(v') = CONST(\top) \wedge v \bowtie_L v'\}$

4. $\forall\{v_1, \ldots, v_k\} \subseteq V_{\mathfrak{o}}$ *with* $h_r(v_i) = h_(v_1)$ *for all* $1 \leq i \leq k$ *such that* $\exists s \in SPARE_F$ *with* $\kappa(v_1) = \sigma(s)_1$, *then for* $2 \leq i \leq k$, $\theta(\kappa(v_i)) \cap SPARE_F = \emptyset$.

Before we present the proof for the theorem, let us shortly review the conditions.

We exclude any spare-gates from the right hand-side to simplify any further arguments, furthermore, we do not allow more than two modules in the left-hand side of the rule, again to simplifiy the argument.

The first condition is regarding the element hierarchy. This condition ensures acyclicity of the resulting DFT. First, we observe that only if the hierarchy relation of elements is extended, cycles can occur. For rules which do not extend the relation, we do not have to put context restrictions. Those rules which change the relation can safely be applied, if it is ensured that after merging everything but one edge, adding this last edge does not contradict the element hierarchy.

The second condition likewise covers module relations. Again, either the rule is module conservative, meaning that no components are merged by the application of the rule. Otherwise, additional context restrictions are necessary to ensure the well-formedness of the resulting DFT. Here, we have to assure that after adding the new connections, we have at most one component which is the result of merging components. This module was either already a single module in the context of the host DFT, or one of the modules was not yet connected to either a spare representant or the top-level element, or the representants are also merged. Furthermore, it assures that the merged module does not contain given failure elements, which is a very restrictive way to ensure that no given failure is added to primary module.

Remark 32. A lesser restriction would take into account that by applying the rules, also some elements are split from their component, which could then be merged with other components. Although such rules are certainly possible, we choose to exclude such scenarios here, in favour of a simpler criterion.

The third criterion prevents adding constant failure elements in the primary component by the right-hand side of the rule.

The fourth criterion ensures that primary components are never shared. We notice that sharing can only be done via the representant, so we only need to restrict the potential predecessors of such representants.

Notice that the restrictions directly yield context restrictions for each rule. We are now ready to present the proof.

Proof sketch of Theorem 5.5. We go through the conditions from Definition 4.9[1].

- The second child of an FDEP is always a basic event.

  Let us assume that the second child of an FDEP is a new element. But then, the FDEP is either new or glued, it is certainly in R. As the rule has to be well-formed, we can be sure that the second child is a basic element. Now, let us assume that the second child of an FDEP is an old element. Likewise, the FDEP is either old or glued, and certainly in F. We require F to be well-formed, so the second child is a basic element. Now, we assume that the second child of an FDEP is a glued element. If the FDEP would be new, then the second child is guaranteed to be a basic element by the well-formedness of the rule. So, the FDEP is original. By the well-formedness of F, the second child was originally a basic element. Therefore, it had no successors and thus was part of the input nodes. For input-nodes, the type does not change, so it is a basic element in F' as well.

- FDEPs have no incoming edges.

  If the FDEP is new, then by well-formedness of the rule, it has no incoming edges. If it is old, it has no incoming edges by the well-formedness of F. For the glued elements, it cannot be in the output-interface by the third restriction. Thus, it is in the input-interface. But then it was also an FDEP in F, so by well-formedness of F, there are no original incoming edges, and by the well-formedness of $\mathfrak{r}$ there are no new incoming edges.

- FDEPs have exactly two children.

  New and old FDEPs are covered by the well-formedness like above. Glued elements are covered as well, as either it is part of the input-interface and no children are either added or deleted by the rule, or it is part of the output-interface, but then all children are in R and by well-formedness of $\mathfrak{r}$, there are exactly two.

- The DFT is acyclic, i.e. $(V, E(\sigma))$ is acyclic.

  Assume F' is cyclic. Then there exist elements v, v' such that $v \in \sigma^*(v')$ and $v' \in \sigma^*(v)$.

[1] albeit in a different order

Let $v_0 \dots v_k$ be a path from v to v' and $u_0 \dots u_l$ be a path from v' to v.

As $\mathfrak{r}$ and F are well-formed, on these pathes there are old and new elements. The cyclic path $v_0 \dots (v_k = u_0) \dots u_l$ thus enters and leaves old elements at least once.

Either $\mathfrak{r}$ is hierarchy conservative. W.l.o.g. v and v' are glued elements with v_j old elements for $1 \leq j < k$. Thus, $v \prec_F v'$. Then for each $u_s \dots u_t$ such that, for all $s \leq i \leq t$, u_i are new elements, there exists an alternative path from u_s to u_t via original nodes (nodes in F). This is a contradiction to the fact that F was acyclic.

Otherwise, the context restriction makes sure that we do not introduce cycles. First, under the assumption that a cycle in F' is introduced, there exists a cycle such that if it enters and leaves new elements at w_i and w'_i respectively, then $w_j \prec_F w'_j$ for all but one pair w_j, w'_j.

Assume there exist w_i, w'_i different from w_j, w'_j with $w_i \not\prec_F w'_i$. W.l.o.g. let $w'_j \prec w_i$. By the restrictions from the theorem, we have that either $w'_j = w'_i$, but then there is a cycle $w'_j \dots w_i \dots w'_i = w'_j$ in F'. Otherwise, we have that there exists a w such that $h_r(w_i) \prec_R w$ and $w \prec_R w'_j$, which would then lead to the cycle $w_i \dots w \dots w'_j \dots w_i$ in F'.

Thus, w.l.o.g. assume v and v' to be glued elements with v_j new elements for $1 \leq j < k$ and $\kappa(v) \not\prec_F \kappa(v')$. We have that $\kappa(v') \not\prec_F \kappa(v')$, thus no cycle exists.

- Extended modules EM_r are independent and sharing is only done via the representant.

  Throughout this proof, we assume that if the rule merges modules the merged modules yield a single spare module in F', which is the precondition of condition 2b in the theorem. First of all, if v, v' as in the precondition do not exist, then no two modules are merged and the rule is module conservative (and we are done directly). Thus, such v, v' exist. Now for all other pair of modules, either they are not merged by the rule, or they are merged with the module generated merging the modules of v, v'.

  We want to show that for each module representant r in F', there does not exists a module path $p \in \mathrm{spmp}_{F'}(r, r')$ with r' either top or another spare module representant, $r' \in \mathsf{EMR}_{F'} \setminus \{r\}$.

  Assume for a contradiction that such a path exists, and let r and r' be the representants which are connected. By the wellformedness of both F and r, we have that the path $rv_1 \dots v_k r'$ consists of both old and new elements.

  - Let us first assume that r and r' are original.

    If $\mathfrak{r}$ is module conservative, then r and r' are only in the same module if they already were in F, which contradicts the wellformedness of F. Else, if $\mathfrak{r}$ merges only modules which were not already in the same module, then we contradict 2(b)i.

    Thus, in F, $r \not\bowtie_F r'$. To assure that $r \not\bowtie'_F r'$, we also have to ensure that for $v, v' \in V_{\mathfrak{o}}$ with $r = \kappa(v)$ and $r' = \kappa(v')$, $h_r(v) \neq h_r(v')$. By condition 2(b)iii, the representants are not merged. Now, we consider the elements $\{x_1, x_2\} \subseteq V_{\mathfrak{i}} \cup V_{\mathfrak{o}}$ which share their module relation only in the rule, not in F. Let r and r' be the corresponding module representant, respectively. On the path $rv_1 \dots v_k r'$, there exist two nodes v_i and v_j with $1 \leq i \leq j \leq k$ with $v_i = \nu'(\kappa(w))$ and $v_j = \nu'(\kappa(w'))$ for some $\{w, w'\} \subseteq V_{\mathfrak{i}} \cup V_{\mathfrak{o}}$ with $w \not\bowtie_L w'$ and $h_r(w) \bowtie_R h_r(w')$. Moreover, we have that $\kappa(w) \bowtie_F r$ and $\kappa(w') \bowtie_F r'$. However, condition 2(b)ii ensures that if $r \bowtie_F \kappa(w)$, then there can not exist w' s.t. $w' \bowtie_F r'$.

  - Let us now assume that r is new and r' is old or r and r' both new. Then, a spare was added. However this is not allowed by the precondition of the theorem.

- Primary spare modules are never shared

  Let r be a primary spare module representant in F' with $s \in F'_{\text{SPARE}}$, $\sigma(s)_1 = r$. We ensure that $\theta(r) = \{s\}$.

  Assume for contradiction that $\theta(r) = \{s_1, s_2\}$ with $s_1, s_2 \in F'_{\text{SPARE}}$.

  If r is old, then certainly, s_1, s_2 are original, so F was not well-formed. If r is new, then s_1, s_2 are also in R, so $\mathfrak{r}$ is not well-formed.

Thus, r is a glued element (in the output interface). Let $r = \nu'(\kappa(v))$ for some $v \in V_{\mathfrak{o}}$. Now, it follows w.l.o.g. that either s_1 is glued and s_2 is old, or that $r = \nu'(\kappa(v'))$ for some $v' \in V_{\mathfrak{o}}$, $v \neq v'$ and $\sigma(s_1) = \kappa(v)$ and $\sigma(s_2) = \kappa(v')$.

First, consider s_1 is glued and s_2 is old. As $h_r(r)$ is a child of $h_r(s_1)$, s_1 is in the output-interface. However, as $R_{\text{SPARE}} = \emptyset$, s_1 cannot be a spare then, which is a contradiction.

Second, consider $r = \nu'(\kappa(x))$ with $x \in \{v_1, v_2\}$. By Condition 4, either s_1 or s_2 are not spare gates, which is also a contradiction.

- Primary spare modules do not contain given-failure elements.

  We want to show that for all $r \in \mathsf{EMR}_{F'}$ and $c \in F'_{\text{CONST}}(\top)$ it holds that $\text{spmp}(r, c) = \emptyset$. Assume for contradiction that there exist such r, c with $\text{spmp}(r, c) \neq \emptyset$.

  Either c is new. As r is original, we have that there must be a path from c to some element $h_r(x)$ with $x \in V_{\mathfrak{i}} \cup V_{\mathfrak{o}}$. However, by Condition 3, then x was already connected to some $c' \in F_{\text{CONST}}(\top)$. This contradicts the wellformedness of F.

  Otherwise c is old, then by the wellformedness of F it was in a module which was merged by the application of the rule. This contradicts Condition 2(b)ii. Therefore, no such c exists.

The facts that exactly the leaf-types have no successors and the threshold of the voting gate is lower than the number of successors follow directly from the well-formedness of F and $\mathfrak{r}$. □

5.2.4. Preserving semantics

Some rewrites change the semantics of a fault tree. We are only interested in rules which are conservative w.r.t. the earlier defined quantitative measures.

Definition 5.12 (Valid rewrites)**.** A rewrite rule $\mathfrak{r}$ is *valid*, if for all well-formed DFTs F, F', such that if rewriting F with $\mathfrak{r}$ using κ yields F', it holds that $F \approxeq F'$. ■

We call a valid rewrite rule $\mathfrak{r} = (L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \mathfrak{C})$ *symmetric*, if $(R \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow L, \mathfrak{C})$ is also a valid rewrite rule.

Removing BEs The *basic events of* $\mathfrak{r}$ are the basic events which occur in L (and therefore in R).

Lemma 5.6. *Given a rewrite rule r and a DFT F such that F is rewritten by $\mathfrak{r}$ to F', $F_{BE} \subseteq F'_{BE}$.*

Proof. Assume that not, then there exists a node $v \in F_{\text{BE}}$ s.t. $v \notin F'_{\text{BE}}$, i.e. $v \in V \wedge \text{Tp}(v) = \text{BE}$ and $v \notin V'$. We rewrite V' and get $v \notin V \setminus \kappa(V_L \setminus (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}))$, thus $v \in \kappa(V_L \setminus (V_{\mathfrak{i}} \cup V_{\mathfrak{o}})) = v \in \kappa(V_L) \setminus (\kappa(V_{\mathfrak{i}}) \cup \kappa(V_{\mathfrak{o}}))$. It remains to show that $v \in V_L \implies v \in \kappa(V_{\mathfrak{i}})$, as that invalidates our assumption. □

As removing of basic events does not fit in our framework, we give a seperate rewrite rule for rewriting unconnected basic events to given failure events.

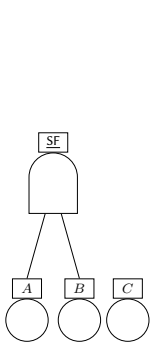
Theorem 5.7. *Let $F = (V, \sigma, Tp, \Theta, top)$ be a well-formed DFT and $v \in F_{BE}$ an basic element such that $\theta(v) = \emptyset$ and $v \neq top$. Then it holds that $F \equiv F'$ for*

$$F' = (V, \sigma, Tp \setminus \{v \mapsto BE\} \cup \{v \mapsto CONST(\bot)\}, \Theta|_{\Theta^{-1}(V \setminus \{v\})}, top)$$

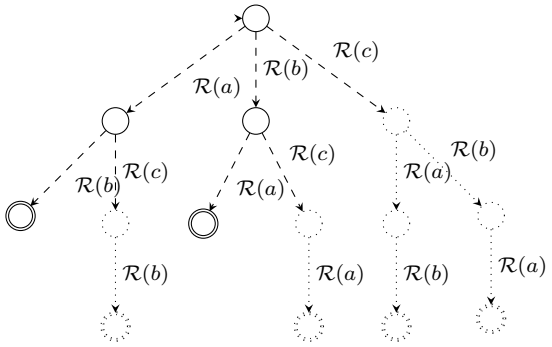
We first illustrate that this should hold by an example.

Example 5.6. In Figure 5.9a, we depict a simple DFT where C is a dispensable basic event. The DFT fails if both A and B have failed. In Figure 5.9b, we depict the corresponding Markov automaton. We omitted the state-labelling for space reasons and dotted the states which are reached after a the occurrence of c. Removal of the basic event C yields a DFT F' where $\mathcal{C}^*_{F'}$ is isomorphic to the continuously drawn part of the figure.

To see why $\text{Rely}_F(t) = \text{Rely}_{F'}(t)$, consider, e.g., the location x reached after the component failure a, $x = (a, A)$. We see that the component failure c in $\mathcal{C}^*_F$ brings us to location (ac, AC) which is isomorphic to the location $x \in \mathcal{C}^*_{F'}$. However, location x differs in the two computation trees. Whereas in $\mathcal{C}^*_{F'}$, component b is the only component participating in the Markovian race, in



(a) A small DFT with an dispensable BE.



(b) The corresponding MA.

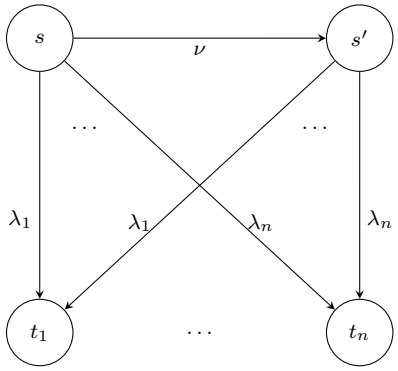


Figure 5.10.: A subchain found in the computation tree of DFT F from Theorem 5.7

$\mathcal{C}_F^*$ also c participates, with two possible outcomes. Either c loses the race, and b fails first. This is exactly the aforementioned scenario. Otherwise, c wins - but with the only effect that we change the location. As Markov automata are memoryless and the failure rates are location-invariant, we conclude that we can safely assume that b wins the race. A similar argument can be made for the location (b, B) where a can be assumed to win over c. This allows us to eliminate the subtrees emerging from the locations (ac, AC) and (bc, BC). After the elimination of these locations, we can apply the same argument for $(\varepsilon, \varepsilon)$, where we can assume that the race is either won by a or by b, but not by c. ▲

The argumentation from the example above is partly captured in the following lemma, which formally shows the conservation of reliabilty in a CTMC (Figure 5.10) closely related to the substructure we find during the removal of the transitions corresponding to a dispensable basic event.

Lemma 5.8. *Let $\mathcal{C} =$ be a CTMC with $S = \{s, s', t_1, \dots t_n\}$ and $R(s, t_i) = R(s', t_i) = \lambda_i$ and $R(s, s') = \nu$ and let all other entries of R be zero. It holds that*

$$\Pr(s \models \Diamond^{\leq t}\{t_i\}) = \Pr(s' \models \Diamond^{\leq t}\{t_i\}) \quad \forall 1 \leq i \leq n$$

Proof. Let $E = \sum_{i=1}^{n} \lambda_i$. The only path from s' to t_i is by the direct transition. Therefore,

$$\Pr(s' \models \Diamond^{\leq t}\{t_i\}) = \frac{\lambda_i}{E} \cdot (1 - e^{-Et}).$$

There exist two pathes from s to t_i: directly or via s'.

$$\Pr(s \models \Diamond^{\leq t}\{t_i\}) = \frac{\lambda_i}{E+\nu} \cdot (1 - e^{-(E+\nu)t}) + \int_0^t \nu \cdot e^{-(E+\nu)x} \cdot \Pr(s' \models \Diamond^{\leq (t-x)} t_i) dx$$

We thus have to show:

$$\frac{\lambda_i}{E} \cdot (1 - e^{-Et}) - \frac{\lambda_i}{E+\nu} \cdot (1 - e^{-(E+\nu)t}) = \int_0^t \nu \cdot e^{-(E+\nu)x} \cdot \Pr(s' \models \Diamond^{\leq (t-x)} t_i) dx$$

Substitution and expansion yields the following for the right hand side:

$$\frac{\nu \cdot \lambda_i}{E} \cdot \int_0^t e^{-(E+\nu)x} - e^{-\nu x} \cdot e^{-Et} dx$$

Using distribution over the integral and solving it yields exactly the left hand side. □

Remark 33. Notice that the lemma also follows from the fact that *weak bisimulation on CTMCs* preserves reachability properties. However, weak bisimulation on Markov automata is not conservative w.r.t. weak bisimulation on CTMCs.

Proof sketch of Theorem 5.7. First, we notice that F' is well-formed. Now, we show that $F' \equiv F$. Consider any location (ρ, π) in the functional transducer of F with $v \notin \rho$. With Corollary 4.24 we follow that

$$\text{Failed}(\pi) \cup \{v\} = \text{Failed}(\pi \cdot v),$$

and thus also $\text{Failable}(\pi) = \text{Failable}(\pi \cdot v)$ and $\text{ClaimedBy}(\pi) = \text{ClaimedBy}(\pi \cdot v)$. This includes $\pi = \varepsilon$. Consequently, using an induction over the length of $\pi' \in \text{BE}^{\triangleright}$.

$$\text{Failed}(\pi \cdot \pi') \cup \{v\} = \text{Failed}(\pi \cdot v \cdot \pi')$$

Moreover, as v is not the dependent event of any FDEP, we have for all reachable locations that $v \in \pi \iff v \in \rho$. On all locations (ρ, π) with $v \notin \rho$, either there exists an outgoing transition (v, v) or $\Delta_\pi \neq \emptyset$.

Let $\mathcal{C}_F = (S, \iota, \text{Act}, \hookrightarrow, \dashrightarrow, \text{AP}, \text{Lab})$ be the MA under F. As $\text{Failed}(\pi \cdot \pi') \cup \{v\} = \text{Failed}(\pi \cdot v \cdot \pi')$ and $v \neq \text{top}$, for $s = (\rho \cdot \rho', \pi \cdot \pi'), s' = (\rho \cdot v \cdot \rho', \pi \cdot v \cdot \pi') \in S$ it holds that $\text{top} \in \text{Lab}(s) \iff \text{top} \in \text{Lab}(s')$.

For any s, s' with $\text{top} \in \text{Lab}(s)$ and $\text{top} \in \text{Lab}(s')$, it holds that $s \approx_s s'$. Moreover, for any s, s' with $F_{\text{BE}} \subseteq \text{Lab}(s)$ and $F_{\text{BE}} \subseteq \text{Lab}(s')$, $s \approx_s s'$.

We will now construct a finite chain

$$\mathcal{C}_F = \mathcal{M}_0 \dots \mathcal{M}_k = \mathcal{C}_{F'}$$

such that $\mathcal{M}_i \equiv \mathcal{M}_{i+1}$ for each $0 \leq i < k$.

We define $k = |F_{\text{BE}}|$.

Let $\mathcal{M}_1 = (S_1, \iota_1, \text{Act}_1, \hookrightarrow_1, \dashrightarrow_1, \text{AP}_1, \text{Lab}_1)$ where all states s with $\text{top} \in \text{Lab}(s)$ and all states with $\text{Lab}(s) \subseteq F_{\text{BE}} \setminus \text{top}$ are merged. We have that $\mathcal{M}_1 \equiv \mathcal{M}_0$.

Now, we consider $\mathcal{M}_i$, $\mathcal{M}_{i+1}$ for $1 \leq i \leq k$. We define $\mathcal{M}_{i+1}$ using $\mathcal{M}_i = (S_i, \iota_i, \text{Act}_i, \hookrightarrow_i, \dashrightarrow_i, \text{AP}_i, \text{Lab}_i)$, as $\mathcal{M}_{i+1} = (S_i, \iota_i, A_i, \hookrightarrow_{i+1}, \dashrightarrow_{i+1}, \text{AP}_i, L_i)$.

Before we define the transition relation, we define some auxiliary constructs. Let $m = k - i$. In $\mathcal{M}_i$, we consider the set $X = X^i_+ \cup X^i_- \subseteq S_i$ where X^i_+ contain all states (ρ, π) with $|\pi| = m$ and $v \notin \pi$ and X^i_- contains all states $|\pi| = m + 1$ and $v \in \pi$.

The transition relation of $\mathcal{M}_{i+1}$ is now given as:

$$\begin{aligned}
\hookrightarrow_{i+1} \quad =& (\hookrightarrow_i \setminus \{((\rho,\pi), a, 1, (\rho, \pi \cdot w)) \mid (\rho,\pi) \in X_+, a \in A_i, w \in F_{\text{BE}}\}) \cup \\
& \{((\rho, \pi \cdot v \cdot \pi'), a, 1, (\rho, \pi \cdot \pi' \cdot w)) \mid \\
& ((\rho, \pi \cdot v \cdot \pi'), a, 1, (\rho, \pi \cdot v \cdot \pi' \cdot w)) \in \hookrightarrow_i \wedge (\rho, \pi \cdot v \cdot \pi') \in X_+\} \\
\dashrightarrow_{i+1} \quad =& (\dashrightarrow_i \setminus \\
& (\{((\rho,\pi), \lambda, (\rho \cdot w, \pi \cdot w)) \mid (\rho,\pi) \in X_+, \lambda \in \mathbb{R}_{>0}, w \in F_{\text{BE}}\} \cup \\
& \{((\rho,\pi), \lambda, (\rho \cdot v, \pi \cdot v)) \mid (\rho,\pi) \in X_-, \lambda \in \mathbb{R}_{>0}\})) \\
& \cup \{((\rho, \pi \cdot v \cdot \pi'), \lambda, (\rho \cdot w, \pi \cdot \pi' \cdot w)) \mid \\
& \quad ((\rho, \pi \cdot v \cdot \pi'), \lambda, (\rho \cdot w, \pi \cdot v \cdot \pi' \cdot w)) \in \dashrightarrow_i \wedge (\rho, \pi \cdot v \cdot \pi') \in X_+\}
\end{aligned}$$

For any $(x, s) \in X_- \times S_{i+1}$ and $(x', s) \in X_+ \times S_{i+1}$ with $x = (\rho, \pi \cdot \pi')$ and $x' = (\rho', \pi \cdot v \cdot \pi')$ the following invariants hold by construction:

1. $(x, a, 1, s) \in \hookrightarrow_{i+1} \iff (x', a, 1, s) \in \hookrightarrow_{i+1} \quad a \in A_{i+1}$
2. $(x, \lambda, s) \in \hookrightarrow_{i+1} \iff (x', \lambda, s) \in \hookrightarrow_{i+1} \quad \lambda \in \mathbb{R}_{>0}$

Next, we show $\mathcal{M}_i \equiv \mathcal{M}_{i+1}$. Assume to the contrary that $\mathcal{M}_i \not\equiv \mathcal{M}_{i+1}$. By using the invariants (1, 2) for on $\mathcal{M}_j$ with $j < i$, we have that for any $x = (\rho, \pi \cdot \pi'), x' = (\rho', \pi \cdot v \cdot \pi'$ with $x \in X_-^j$ and $x' \in X_+^j$,

$$\text{outgoing}^{\mathcal{M}_i}(x) = \text{outgoing}^{\mathcal{M}_i}(x'),$$

and thus

$$\overset{\mathcal{M}_i}{\Pr}(x \models \Diamond^{\leq t}\text{top}) = \overset{\mathcal{M}_i}{\Pr}(x' \models \Diamond^{\leq t}\text{top}) \quad \forall t \in \mathbb{R}_{\geq 0}.$$

Moreover, as no outgoing connection is changed for states x with $x \in X^j$ with $j \neq i$, we have

$$\text{outgoing}^{\mathcal{M}_i}(x) = \text{outgoing}^{\mathcal{M}_{i+1}}(x),$$

and thereby for $x \in X^j, j < i$

$$\overset{\mathcal{M}_i}{\Pr}(x \models \Diamond^{\leq t}\text{top}) = \overset{\mathcal{M}_{i+1}}{\Pr}(x \models \Diamond^{\leq t}\text{top}) \quad \forall t \in \mathbb{R}_{\geq 0} \tag{5.1}$$

and for $x \in X^j, j > i$

$$\overset{\mathcal{M}_i}{\Pr}(x \models \Diamond^{\leq t}\text{Lab}(y)) = \overset{\mathcal{M}_{i+1}}{\Pr}(x \models \Diamond^{\leq t}\text{Lab}(y)) \quad \forall t \in \mathbb{R}_{\geq 0}, y \in X^i. \tag{5.2}$$

It remains to show that

$$\overset{\mathcal{M}_i}{\Pr}(x \models \Diamond^{\leq t}\text{Lab}(y)) = \overset{\mathcal{M}_{i+1}}{\Pr}(x \models \Diamond^{\leq t}\text{Lab}(y)) \quad \forall t \in \mathbb{R}_{\geq 0}, y \in X^{i-1} \tag{5.3}$$

We distinguish two cases, x Markovian and x interactive.

- x interactive.

  For $x \in X_-^i$, we already have $\text{outgoing}^{\mathcal{M}_i}(x) = \text{outgoing}^{\mathcal{M}_{i+1}}(x)$ and by (5.1), (5.3) follows.

  For $x \in X_+^i$, we change the outgoing transitions. However, for each $a \in A_i$ and $y \in S_i$, for each $(x, a, 1, y) \in \text{outgoing}^{\mathcal{M}_i}$ that is removed, a new transition $(x, a, 1, y')$ is added. By (5.2) we have that $\text{outgoing}^{\mathcal{M}_i}(y) = \text{outgoing}^{\mathcal{M}_i}(y')$.

- x Markovian. The situation matches the construct handled in Lemma 5.8. The lemma indeeds directly yields the validity of (5.3).

Now, it remains to show that $\mathcal{M}_k \equiv \mathcal{C}^*_{F'}$. This is intuitively clear, as we removed all states (and transitions to such states) where the removed basic event failed. We did not touch any other states or transitions. We omit a technical proof of this statement. Notice that $\mathcal{M}_k$ indeed possesses a richer label set. For the requested equivalence modulo the measures of interest, this is not of importance.

We thus have a finite chain of Markov automata where each adjacent pair has equivalent measures of interest. The first MA is equivalent to the computation tree of the original DFT and the last is equivalent to the resulting DFT. By transitivity, the DFTs are thus weakly equivalent. □

We have another rule of great practical relevance which does not directly fit in the framework. The basic idea is depicted in Figure 5.11. Two basic events with an attached failure rate of $\mathcal{R}(\omega_1)$ and $\mathcal{R}(\omega_2)$, respectively, can be merged to a single basic event with an attached component failure with a failure rate $\mathcal{R}(\omega_1) + \mathcal{R}(\omega_2)$, but only if the original events are not connected to any other gates. Intuitively, we then introduce a single event which encodes the failure of the or-gate. The failure of the or-gate is given by the Markovian race of all its children.

The correctness of the approach is formalised in the next theorem.

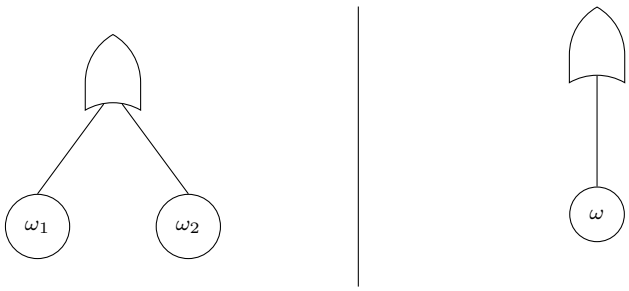


Figure 5.11.: Merging two basic events which are connected only to the same or-gate. The rate of ω is given in Theorem 5.9

Theorem 5.9. *Let $F_\Omega = (V, \sigma, Tp, \Theta, top)$ be a well-formed DFT and $v \in F_{OR}$ such that $\sigma(v) = \{v_1, v_2\} \subseteq F_{BE}$ and $v_2 \neq top$. Let $\theta(v_1) = \theta(v_2) = \{v\}$ and $\{\omega_1, \omega_2\} \subseteq \Omega$ with $\Theta(\omega_i) = v_i$ for $i \in \{1, 2\}$. Then for $F'_{\Omega'} = (V', \sigma', Tp', \Theta', top')$ with*

$$\Omega' = \Omega \setminus \{\omega_1, \omega_2\} \cup \{\omega\} \text{ s.t. } \mathcal{R}(\omega) = \mathcal{R}(\omega_1 + \omega_2) \text{ and } \alpha_\omega = \frac{\alpha_{\omega_1}\mathcal{R}(\omega_1) + \alpha_{\omega_2}\mathcal{R}(\omega_2)}{\mathcal{R}(\omega)}$$

and

- $V' = V \setminus \{v_2\}$,
- $\sigma' = \sigma_{|V'} \setminus \{v \mapsto v_1 v_2\} \cup \{v \mapsto v_1\}$,
- $Tp' = Tp_{|V'}$,
- $\Theta = \Theta_{|\Omega'} \cup \{\omega \mapsto v_1\}$,
- $top' = top$.

We do not give the proof here. A further generalisation of the framework is thus required here. Although not in the scope of this thesis, we shortly discuss how the framework could be extended. First of all, we observe that a (deterministic) DFT encodes a phase-type distribution. Obviously, there exist several DFTs which encode the same phase-type distribution, see the theorem above for an example. Subtrees also encode a phase-type distribution, given by the DFT which is isomorphic to the subtree. Such a subtree can thus be replaced by another DFT which encodes the same phase type distribution without affecting the host DFT.

5.3. Correctness of rewrite rules

In the previous sections, we discussed in detail why rules can be context sensitive. We saw earlier that each rewrite rule has a corresponding set on which it should not be applied.

Instead of defining these DFTs explicitly for each rule, we introduce a small collection of contexts, that is, DFTs which allow the valid application of a set of rules. It is important to notice that often, the context restriction could be weakened considerably. We however choose to present the rather strong restrictions as this eases understanding. For each context restriction, we develop a *proof obligation*, a criterion on the rule that suffices to proof its validity. This notably streamlines the later presentation of rewrite rules, as for each of these rules, we merely have to show that the rules fulfill these proof obligation.

5.3.1. Validity of rules without FDEPs and SPAREs

We start with a couple of contexts for rules which do not contain failure forwarding, claiming or activation. Notice that this not mean that these mechanisms are not present in the remainder of the DFT. We only exclude the elements from being matched or added here. Later, we add functional dependencies and failure forwarding to the rules we consider.

We define the auxiliary concept of failed under an oracle. Intuitively, we have no or only partial knowledge of the context of the DFT. As we want equal behaviour for many[1] states, we ask an

[1] To see why we write many, instead of all, consider an and-gate with the top-level element of a DFT as successor.

oracle to tell us the state of the input nodes of our rewrite rule. Based on the oracle, the failure is then propagated through the subDFTs. This propagation works exactly as before.

Definition 5.13. Let $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with basic events B and let $B' \supseteq B$. We call a function f, $f\colon \mathcal{P}(V_\mathrm{o}) \times B'^{\triangleright} \rightarrow \mathcal{P}(V_\mathrm{i})$ such that $f(X,\pi) \subseteq f(X \cup Y, \pi)$ and $f(X,\pi) \subseteq f(X, \pi \cdot x)$ an *input-oracle for* $\mathfrak{r}$. If $f(X,\pi) = f(Y,\pi)$ for all $X, Y \subseteq V_\mathrm{o}$, then we call f *output-independent.* ■

In the following, we restrict ourselves to output-independent oracles. We omit passing the state of output nodes and write $f\colon B'^{\triangleright} \rightarrow \mathcal{P}(V_\mathrm{i})$.

Definition 5.14. Let $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with basic events B and let $B' \supseteq B$, and f an oracle for $\mathfrak{r}$. Let $\pi \in B'^{\triangleright}$ and $F \in \{L, R\}$ a subDFT with elements V_F. We recursively characterise the set $\mathrm{Failed}_F[f](\pi)$ of *failed elements considering* f. For $v \in \kappa(V_\mathrm{i})$, we have that $v \in \mathrm{Failed}_F[f](\pi)$ iff $v \in f(\pi)$. For $v \in V_F \setminus \kappa(V_\mathrm{i})$, we have that $v \in \mathrm{Failed}_F(\pi)$ iff $v \models_F \pi$ given that $v' \models_F \pi$ for all $v' \in f(\pi)$[1]. ■

A large set of rules can actually be applied without any restrictions, other than those necessary for valid syntax, on the context. Typical examples of this class of rules are rules such as flattening.

Theorem 5.10 (Unrestricted context proof obligation). *Let $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with basic events B and let $B' \supseteq B$. Let $L_{FDEP} = L_{SPARE} = R_{FDEP} = R_{SPARE} = \emptyset$. Then $\mathfrak{r}$ is valid if*

$$\forall \pi \in B'^{\triangleright}\ \forall f \text{ oracle for } \mathfrak{r}$$
$$h_r(\mathit{Failed}_L[f](\pi) \cap V_\mathrm{o}) = \mathit{Failed}_R[f](\pi) \cap h_r(V_\mathrm{o})$$

and for all $\{v, v'\} \subseteq V_\mathrm{i} \cup V_\mathrm{o}$ it holds that

$$v \bowtie_L v' \iff h_r(v) \bowtie_R h_r(v').$$

For the proof, we transfer the oracle function to the host DFT and the result. We omit a rigorous definition and refer to Definition 4.12.

Definition 5.15. Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ with oracle f and DFTs F, F' such that $F \xrightarrow{r,\kappa} F'$. The set $\mathrm{Failed}_F[f](\pi)$ is recursively defined. For $v \in \kappa(V_\mathrm{i})$, we define $v \in \mathrm{Failed}_F[f](\pi)$ iff $v \in \{\kappa(v') \mid \exists v' \in f(\pi)\}$. For $v \in F \setminus \kappa(V_\mathrm{i})$, we define $v \in \mathrm{Failed}_F[f](\pi)$ iff $v \models_F \pi$, given that $v' \models_F \pi$ for all $v' \in \kappa(f(\pi))$ and $v' \not\models_F \pi$ for all $v' \in \kappa(V_\mathrm{i} \setminus f(\pi))$. Furthermore, we define $\mathrm{Failed}_{F'}[f](\pi)$ analogously. For $v \in \eta(h_r(V_\mathrm{i}))$, we define $v \in \mathrm{Failed}_{F'}[f](\pi)$ iff $\{\eta(h_r(v)) \mid v \in f(\pi)\}$. For $v \in F' \setminus \eta(h_r(V_\mathrm{i}))$, we define $v \in \mathrm{Failed}_{F'}[f](\pi)$ iff $v \models_F \pi$, given that $v' \models_F \pi$ for all $v' \in \eta(h_r(f(\pi)))$ and $v' \not\models_F \pi$ for all $v' \in \eta(h_r(V_\mathrm{i} \setminus f(\pi)))$. ■

Proof of Theorem 5.10. Let $F = (V, \sigma, \mathrm{Tp}, \Theta, \mathrm{top})$ be an arbitrary DFT with $B' = F_\mathrm{BE}$, such that $\mathfrak{r}$ matches with κ on F. Let $F' = (V', \sigma', \mathrm{Tp}', \Theta', \mathrm{top}')$ be the result of rewriting, i.e. $F \xrightarrow{r,\kappa} F'$. As we only want to show validity, we can assume that F' is well-formed.

We start by showing F and F' are configuration-equivalent.

1. W.l.o.g. we assume that $F_\mathrm{BE} = F'_\mathrm{BE}$.

   The easy direction is $F_\mathrm{BE} \subseteq F'_\mathrm{BE}$ and is covered by Lemma 5.6. We want to show that $F'_\mathrm{BE} \subseteq F_\mathrm{BE}$. Now assume that $v \in F'_\mathrm{BE}$ and $v \notin F_\mathrm{BE}$. Then, it follows that $v \in V_R \setminus (V_\mathrm{i} \cup V_\mathrm{o})$. Furthermore, as F' uses the attachment function from F[2], we have that every such v must be a dummy event. As we do not allow any FDEP gates to be added, and $v \notin V_\mathrm{i}$, v never fails in F' and can be replaced elements with type $\mathrm{CONST}(\bot)$.

2. We notice that $\theta^*(\eta(R)) \setminus \eta(R) = \theta^*(\eta(h_r(V_\mathrm{o}))) \setminus \eta(R)$, which follows directly from the DPO construction. As we have $\theta^*(\eta(R)) \setminus \eta(R) \subseteq \nu'(D \setminus (V_\mathrm{i} \cup V_\mathrm{o}))$, it is safe to say that $\theta^*(\eta(R)) \setminus \eta(R) = \theta^*(\kappa(L)) \setminus \kappa(L)$. Let Y be this set of elements. Obviously, $Y \cap F_\mathrm{BE} = \emptyset$.

[1] We omit a rigorous definition here, instead we refer to Definition 4.12.

[2] We are not rewriting any of the nodes encoding the attachment function

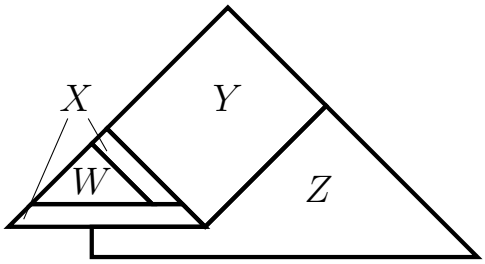


Figure 5.12.: Illustration for the proof of Theorem 5.10.

3. We build a partition of V' such that:

   $$V' = W' \cup X \cup Y \cup Z$$

   with $W' = \eta(R) \setminus \eta(V_{\mathfrak{i}} \cup V_{\mathfrak{o}}))$ and $X = \eta(h_r(V_{\mathfrak{i}} \cup V_{\mathfrak{o}}))$.

   We notice that all elements in $X \cup Y \cup Z$ are old elements, they are also present in F. Furthermore, for all elements in $Y \cup Z$, all successors are also old elements.

   We sketch the partition in Figure 5.12.

   This yields a similar partition of V:

   $$V = W \cup X \cup Y \cup Z$$

   with $W = \kappa(L) \setminus \kappa(V_{\mathfrak{i}} \cup V_{\mathfrak{o}}))$ and $X = \eta(h_r(V_{\mathfrak{i}} \cup V_{\mathfrak{o}}))$.

4. By induction over the length of π, we show that for all $v \in Z$ and for all $\pi \in B'^{\triangleright}$, $v \in \text{Failed}_F[f](\pi) = v \in \text{Failed}_{F'}[f](\pi)$.

   The base case $\pi = \varepsilon$ follows with structural induction directly, as for all $v \in Z$, $\{z \in V \mid v \prec_F z\} = \{z \in V' \mid v \prec_{F'} z\}$.

   We postpone the induction step for a moment, and first introduce our next claim.

5. We show that for all $v \in Y$ and for all $\pi \in B'^{\triangleright}$, $v \in \text{Failed}_F[f](\pi) \iff v \in \text{Failed}_{F'}[f](\pi)$, again by induction voer π.

   For the base case, we use again a structural argument. Assume $v \in \text{Failed}_F[f](\pi)$ but $v \notin \text{Failed}_{F'}[f](\pi)$. This implies that $v \notin \text{FDEP}$. Then there exists a successor $v' \in \sigma(v)$ such that $v; \in \text{Failed}_F[f](\pi)$ but $v' \notin \text{Failed}_{F'}[f]$.

   We distinguish several cases.

   - $v' \in Y$, but then we can repeat this argument by taking v' as v. As DFTs are acyclic and finite, we have an ascending chain, thus we enter this case only finitely often.
   - $v' \in W'$ is impossible as $\mathfrak{r}$ was matching.
   - $v' \in X$ contradicts the precondition from the theorem.
   - $v' \in Z$ is covered by the base case from the former induction.

   The case $v \notin \text{Failed}_F[f](\pi), v \in \text{Failed}_{F'}[f](\pi)$ is analogous.

   We now do the induction step for $v \in Y$ and $v \in Z$.

   Again, we assume that there exists a node $v \in Y \cup Z$, $v \notin \text{FDEP}$, s.t. $v \in \text{Failed}_F[f](\pi)$ and $v \notin \text{Failed}_{F'}[f](\pi)$.

   We use a structural induction over the graph of the (sub)DFT $X \cup Y \cup Z$.

   For the sinks in the graph, we have $v \in X \cup Z$. The case $v \in X$ is handled by the precondition. We regard $v \in Z$. For $v \in \text{BE}$, the only reason for a difference between F and F' would be if v' is a dependent event, which has been triggered after a $\pi' \in \text{pre}(\pi)$. But then our induction hypothesis interferes.

   For all static gates we can use the structural argument as in the base. For pand-gates, we notice that $\text{Failable}_F(\pi) \cap (Y \cup Z) = \text{Failable}_{F'}(\pi) \cap (Y \cup Z)$, as otherwise, there must be

some w.l.o.g. a $v \in \text{Failable}_F(\pi), v \notin \text{Failable}(\pi)$ with $v' \in \sigma(v)$ and $v' \in \text{Failed}_F[f](\pi')$ and $v' \notin \text{Failed}_{F'}[f](\pi')$ for $\pi' \in \text{pre}(\pi)$, which is ruled out by the induction hypothesis.

It remains to handle $v \in \text{SPARE}$. With Lemmma 4.18, it suffices to show that

$$\text{LastClaimed}_F(\pi_{|-1}, v) = \text{LastClaimed}_{F'}(\pi_{|-1}, v)$$

and

$$\text{Available}_F(\pi, v) = \text{Available}_{F'}(\pi, v)$$

. By applying the definition of LastClaimed and the induction hypothesis, we conclude that this boils down to proving that $\text{Available}_F(\pi', v) = \text{Available}_{F'}(\pi', v)$ for all $\pi' \in \text{pre}(\pi) \cup \{\pi\}$. We show this by a nested induction over the length of π'. The base step is trivial.

So, by the induction hypothesis, we assume $\text{Available}_F(\pi'_{|-1}, v) = \text{Available}_{F'}(\pi'_{|-1})$. There are four reasons why $\text{Available}_F(\pi', v) \neq \text{Available}_{F'}(\pi', v)$. W.l.o.g. we assume $\exists v' \in \text{Available}_F(\pi', v) \setminus \text{Available}_{F'}(\pi', v)$.

- $\exists v' \in \text{Available}_F(\pi'_{|-1}, v) \cap \text{Failed}_F[f](\pi')$ and $v' \notin \text{Failed}_F[f](\pi')$. By the structural induction hypothesis for $\text{Failed}(\pi)$ we can exclude this.
- v claiming after $\pi'_{|-1}$, but then it $\text{LastClaimed}(\pi'_{|-1}, v) \in \text{Failed}(\pi'_{|-1})$, which contradicts the induction hypothesis from the induction over π for $\text{Failed}(\pi)$.
- Another spare $s \in V$ claimed v' after $\pi'_{|-1}$, which contradicts the induction hypothesis. We have that $s \in Y \cup Z$, as with $L_{\text{SPARE}} = R_{\text{SPARE}} = \emptyset$, we have that $s \notin W \cup X$ (for the other direction, we have $s \notin W' \cup X$.

6. We notice that

$$\exists f' \colon B'^{\triangleright} \to \mathcal{P}(V_\text{i}) \text{ s.t. } \text{Failed}_F[f'](\pi) = \text{Failed}_F(\pi) \wedge \text{Failed}_{F'}[f'](\pi) = \text{Failed}_{F'}(\pi)$$

We first observe that the following statement would be trivially true: $\exists f, f' \in B'^{\triangleright} \to \mathcal{P}(V_\text{i})$ such that $\text{Failed}_F[f](\pi) = \text{Failed}_F(\pi) \wedge \text{Failed}_{F'}[f'](\pi) = \text{Failed}_{F'}(\pi)$. That is, we reach for all $\pi \in B'^{\triangleright}$ over all subsets of V_i - thereby we must hit the correct set eventually. So the interesting part is that there is indeed an oracle which fulfils the left and the right side.

Assume for a contradiction that this is not true, then there exists an oracle f and $\pi \in B^{\triangleright}$ with $\text{Failed}_F[f](\pi) = \text{Failed}_F(\pi)$ but $\text{Failed}_{F'}[f](\pi) \neq \text{Failed}_{F'}(\pi)$. We show that $v \in \text{Failed}_{F'}[f](\pi) \setminus \text{Failed}_{F'}(\pi)$ leads to a contradiction, the other direction is analogous. W.l.o.g. we consider a minimal π, i.e. we assume $\forall \pi' \in \text{pre}(\pi)$ that $\text{Failed}_{F'}[f](\pi) = \text{Failed}_{F'}(\pi)$. Moreover, we notice that we only have to consider elements in $\eta(h_r(V_\text{i}))$. If for each $v \in \eta(h_r(V_\text{i}))$, $v \in \text{Failed}_{F'}[f](\pi) \iff v \in \text{Failed}_{F'}(\pi)$, then by Definition 5.15, this holds for all $v \in V$. Let us thus consider such a $v \in \eta(h_r(V_\text{i}))$. We have that $\sigma(v) \subseteq X \cup Y \cup Z$. Following points (4) and (5), we have that any $v' \in \sigma(v)$,

$$v' \in \text{Failed}_{F'}[f](\pi) \iff v' \in \text{Failed}_F(\pi).$$

We only sketch the further proof here — it follows closely the arguments from (4). For any $v' \in Z$, a structural induction (like in (5)) using the fact that all elements are old shows that $v' \in \text{Failed}_{F'}(\pi) \iff v' \in \text{Failed}_F(\pi)$. It follows directly that any for v with $\sigma(v) \subseteq Y$, $v \in \text{Failed}_{F'}(\pi) \iff v' \in \text{Failed}_F(\pi)$. The structural induction can then be extended to any $v \in \eta(h_r(V_\text{o}))$ with $\sigma^*(v) \cap \eta(h_r(V_\text{i})) \subseteq \{v \in \eta(h_r(V_\text{i})) \mid \sigma(v) \in Y\}$ by the precondition of the theorem — and subsequently also to their predecessors.

We thus obtain the following key step in the proof: The exists an oracle f such that for all $v \in F \setminus \kappa(L \setminus (V_\text{i} \cup V_\text{o}))$ and for all $\pi \in B'^{\triangleright}$, $v \in \text{Failed}_F[f](\pi) \iff v \in \text{Failed}_{F'}[f](\pi)$ implies

$$v \in \text{Failed}_F(\pi) \iff v \in \text{Failed}_{F'}(\pi).$$

7. By the requirements and application of rewrite rules, $\text{top} = \text{top}'$ and $\text{top} \in X \cup Y \cup Z$. Thus, $\text{top} \in \text{Failed}_F(\pi) \iff \text{top}' \in \text{Failed}_{F'}(\pi)$.

8. As $L_{\text{FDEP}} = R_{\text{FDEP}} = \emptyset$, we have that $\forall \pi \in B'^{\triangleright} \Delta_F(\pi) = \Delta'_F(\pi)$.

Thus, F and F' have isomorphic (up to the labelling of the nodes) functional transducers.

We continue to show that the computation trees with the restricted label set are also isomorphic, i.e. $\mathcal{C}^*_F \simeq \mathcal{C}^*_{F'}$.

9. F and F' range over the same set of component failures.

10. The attachment function is untouched, i.e. we have $\Theta = \Theta'$.

11. It remains to show that for all $\pi \in B'^{\triangleright}$, we have $\text{Activated}_F(\pi) = \text{Activated}'_F(\pi)$. To this end we show $\text{Active}_F(\pi) \cap (F_{\text{SPARE}} \cup F_{\text{BE}}) = \text{Active}_{F'}(\pi) \cap (F'_{\text{SPARE}} \cup F'_{\text{BE}})$. We know from above that $\text{ClaimedBy}_F(\pi, s) = \text{ClaimedBy}_{F'}(\pi, s')$.

    We can easily show that $\mathsf{EMR}_F = \mathsf{EMR}_{F'}$ as $W \cap \mathsf{EMR}_F = W' \cap \mathsf{EMR}_{F'} = \emptyset$.

    We thus have to show that

    $$\forall v \in \mathsf{EMR}_F, \mathsf{EM}_{F,v} \cap (F_{\text{SPARE}} \cup F_{\text{BE}}) = \mathsf{EM}_{F',v} \cap (F'_{\text{SPARE}} \cup F'_{\text{BE}}).$$

    As $W \cap (F_{\text{SPARE}} \cup F_{\text{BE}}) = \emptyset = W' \cap (F'_{\text{SPARE}} \cup F'_{\text{BE}})$, it suffices to show

    $$\forall v \in \mathsf{EMR}_F, \mathsf{EM}_{F,v} \cap (X \cup Y \cup Z) = \mathsf{EM}_{F',v} \cap (X \cup Y \cup Z).$$

    Assume that not, then w.l.o.g. there exist $v \in \mathsf{EMR}_F$ and a $v' \in X \cup Y \cup Z$ with $v \bowtie_F v'$ and $v \not\bowtie_{F'} v'$, that is, a spare module path was removed by the application of the rule.

    $$\exists p = v_0 e_1 \dots v_n \in \text{spmp}_F(v, v') \; \exists i < n v_i v_{i+1} \in W \cup X$$

    We must enter and leave $W \cup X$, so there exist $i' \leq i$ and $i'' > i$ such that $v_{i'}, v_{i''} \in X$ and $v_j \in W \cup X$ for all $i' \leq j \leq i''$. By the precondition, we have that $v_{i'} \bowtie_L v_{i''} \implies v_{i'} \bowtie_{R'} v_{i''}$. Thus, in F', there is a path p'_i from $v_{i'}$ to $v_{i''}$. Please notice that multiple such path fragments may exist. However, these can all be eliminated likewise.

From Corollary 4.38, we deduce that $F \equiv F'$. □

Although rules have a direction, many rules can be applied in both directions.

Definition 5.16. Given a valid rewrite rule $\mathfrak{r} = (L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \emptyset)$. If $(R \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow L, \emptyset)$ is a valid rewrite rule, then $\mathfrak{r}$ is called *symmetric*. ■

Indeed, rules with an injective h_r homomorphism and without context restriction are always *symmetric*. This is captured formally by the following proposition.

Proposition 5.11. *Given a rewrite rule* $\mathfrak{r} = (L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \emptyset)$. *If* $(V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R$ *is injective, then* $\mathfrak{r}$ *is symmetric.*

The proof follows directly from the symmetric nature of the proof obligation stated in Theorem 5.10.

We now present modified proof obligations for rules which have a context restriction. The first context (restriction) is used to ensure that two elements in a matched subDFT surely never fail after the same basic event. We therefore define the set of graphs where the given two elements may fail simultaneous and add them to the restriction set.

Definition 5.17 (Independent inputs context)**.** Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x, y \in V_{\mathfrak{i}}$. The set $\text{IndependentInputs}(x, y) \subseteq \text{DFTs} \times (L \rightarrow \text{IndependentInputs}(x, y)_1)$ is defined as

$$\{(F, \zeta) \mid \forall p \in \text{ifcp}(\zeta(x)) \forall q \in \text{ifcp}(\zeta(y)) p_{\downarrow} \neq q_{\downarrow}\}$$

A rewrite rule $\mathfrak{r}$ possesses the *independent input context restriction w.r.t.* x *and* y if

$$\text{IndependentInputs}(x, y) \subseteq \mathfrak{C}.$$ ■

With independent inputs, we can assure that two input elements never fail simultaneously. We can therefore restrict the set of oracles.

Theorem 5.12 (IndependentInputs proof obligation). *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with basic events* B *and let* $B' \supseteq B$. *Let* $L_{FDEP} = L_{SPARE} = R_{FDEP} = R_{SPARE} = \emptyset$ *and* $A = \{\{v_1, v_2\} \mid \mathit{IndependentInputs}(v_1, v_2) \subseteq \mathfrak{C}\} \subseteq \mathcal{P}(V_\mathfrak{i})\}$ *Then* $\mathfrak{r}$ *is valid if*

$$\begin{aligned}&\forall \pi \in B'^{\triangleright}\\ &\forall f \text{ oracle for } \mathfrak{r} \text{ s.t. } \forall\, a \in A\, \forall x \in B'\ a \not\subseteq f(\pi \cdot x) \setminus f(\pi)\\ &\quad h_r(\mathit{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \mathit{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})\end{aligned}$$

and for all $\{v, v'\} \subseteq V_\mathfrak{i} \cup V_\mathfrak{o}$ *it holds that*

$$v \bowtie_L v' \iff h_r(v) \bowtie_R h_r(v').$$

Remark 34. Please notice that this does not exclude simultaneous failures (consider $\pi = \varepsilon$).

Corollary 5.13. *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule as in Theorem 5.12, To prove* $\mathfrak{r}$ *is valid, we can assume* $\{v, v'\} \not\subseteq \mathit{JustFailed}_L[f](\pi)$ *and* $\{v, v'\} \not\subseteq \mathit{JustFailed}_R[f](\pi)$ *for all* $\pi \in B'^{\triangleright} \setminus \varepsilon$.

Proof sketch of Theorem 5.12. We only have to adapt Item 6 on page 135 from the proof for Theorem 5.10.

Consider the oracle f which fulfils $\mathrm{Failed}_F(\pi) = \mathrm{Failed}_F[f](\pi)$. Assume that there exist a $\pi \in B'^{\triangleright}$ and $x \in B' \setminus \pi$ such that $\{v_1, v_2\} \subseteq \mathrm{Failed}_F(\pi \cdot x) \setminus \mathrm{Failed}_F(\pi)$, that is $\{v_1, v_2\} \subseteq \mathrm{JustFailed}(\pi \cdot x)$. By Corollary 4.24 we conclude that $\exists p \in \mathrm{ifcp}(v)$ and $\exists p' \in \mathrm{ifcp}(v')$ with $p_\downarrow = x = p'_\downarrow$. This, however is not possible as by the context restriction, there is $z \in V$ with $\exists p \in \mathrm{ifcp}(v) \exists p' \in \mathrm{ifcp}(v')\ p_\downarrow = p'_\downarrow$.

□

Furthermore, we regularly want to assure that an element in a matched subDFT has not failed initially.

Definition 5.18 (Event-dependent context). Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x \in V_\mathfrak{i}$. The set $\mathrm{EventDependentFailure}(x) \subseteq \mathrm{DFTs} \times (L \rightarrow \mathrm{EventDependentFailure}(x)_1)$ is defined as

$$\{(F, \zeta) \mid \exists p \in \mathrm{ifcp}(\zeta(x))\ p_\downarrow \in \mathrm{CONST}(\top)\}$$

A rewrite rule $\mathfrak{r}$ possesses the *event dependent context restriction w.r.t.* x if

$$\mathrm{EventDependentFailure}(x) \subseteq \mathfrak{C}.$$ ■

Theorem 5.14 (EventDependentFailure proof obligation). *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with basic events* B *and let* $B' \supseteq B$. *Let* $L_{FDEP} = L_{SPARE} = R_{FDEP} = R_{SPARE} = \emptyset$ *and* $A = \{v \mid \mathit{EventDependentFailure}(v) \subseteq \mathfrak{C}\} \subseteq V_\mathfrak{i}\}$. *Then* $\mathfrak{r}$ *is valid if*

$$\begin{aligned}&\forall \pi \in B'^{\triangleright}\\ &\forall f \text{ oracle for } \mathfrak{r} \text{ s.t. } \forall\, a \in A\, \forall x \in B'\ a \notin f(\varepsilon)\\ &\quad h_r(\mathit{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \mathit{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})\end{aligned}$$

and for all $\{v, v'\} \subseteq V_\mathfrak{i} \cup V_\mathfrak{o}$ *it holds that*

$$v \bowtie_L v' \iff h_r(v) \bowtie_R h_r(v').$$

Corollary 5.15. *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rule as in Theorem 5.14. To prove that* $\mathfrak{r}$ *is valid, we may assume that* $v \notin \mathit{Failed}_L[f](\varepsilon)$ *and* $v \notin \mathit{Failed}_R[f](\varepsilon)$.

Proof sketch of Theorem 5.14. As in the proof for Theorem 5.12, we adapt the proof of Theorem 5.10 by reconsidering point Item 6 on page 135.

Consider the oracle f which fulfils $\text{Failed}_F(\pi) = \text{Failed}_F[f](\pi)$. Assume that there exist a $v \in \text{Failed}_F(\varepsilon)$ with $v \in \text{EventDependentFailure}$. By Corollary 4.25 we conclude that $\exists p \in \text{ifcp}(v)$ with $p_\downarrow \in \text{CONST}(\top)$. This directly contradicts the context restriction. □

In Example 5.1 on page 113, we saw that we need to assure that rewriting may cause changes in the elements which are activated. Here, we restrict ourselves to problems that may arise if connections are eliminated. Problems do not arise if either

- All basic elements and spares are activated as before, because the interface elements are still connected in the DFT if the matched subDFT is erased.
- The basic events which are disconnected from the representant are (after the application of the rule) not on immediate cause failure paths from triggers of functional dependencies and there are no spare gates disconnected.
- The elements are not in a spare module as they're not connected to a spare or the top-level, therefore, they are never activated.

We therefore define a context restriction which prevents a rule from being applied on any DFT where none of the three properties are fulfilled.

Definition 5.19 (Activation connection context)**.** Let $\mathfrak{r} = (L \leftarrow (V_\text{i} \cup V_\text{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x, y \in V_\text{i} \cup V_\text{o}$. The set $\text{ActivationConnection}(x, y) \subseteq \text{DFTs} \times (L \rightarrow \text{ActivationConnection}(x, y)_1)$ is defined as

$$\{(F, \zeta) \mid \neg(\phi_1(F, \zeta, x, y) \vee \phi_2(F, \zeta, x, y) \vee \phi_3(F, \zeta, x, y))\}$$

with

$$\begin{aligned}
\phi_1(F, \zeta, x, y) \quad &= \exists p \in \text{spmp}_F(\zeta(x), \zeta(y))\ p = v_0 e_1 v_1 \dots e_n v_n \wedge \forall 1 \leq i \leq n\ e_i \notin \zeta(L) \\
\phi_2(F, \zeta, x, y) \quad &= \phi_2'(F, \zeta, x) \wedge \phi_2'(F, \zeta, y) \\
\phi_2'(F, \zeta, z) \quad &= (\forall r \in \text{EMR}_F\ \forall v_0 e_1 \dots v_m = p \in \text{spmp}_F(\zeta(z), r) \exists i \leq m\ e_i \in \zeta(L)) \\
&\qquad \Longrightarrow \\
&\qquad (\forall z' \in F_\text{BE}\ \text{spmp}_F(\zeta(z), z') \neq \emptyset\ \nexists t \in \{v_1 | v \in \text{FDEP}\} \\
&\qquad\quad \exists p \in \text{ifcp}(t)\ p_\downarrow = z' \wedge \forall z' \in F_\text{SPARE}\ \text{spmp}_F(\zeta(z), z') = \emptyset \\
\phi_3(F, \zeta, x, y) \quad &= \nexists z \in \text{EMR}_F\ (\text{spmp}_F(\zeta(x), z) \neq \emptyset \vee \text{spmp}_F(\zeta(y), z) \neq \emptyset)
\end{aligned}$$

A rewrite rule $\mathfrak{r}$ possesses the *activation connection context restriction w.r.t. x and y* if $\text{ActivationConnection}(x, y) \subseteq \mathfrak{C}$ ■

Theorem 5.16. *Let $\mathfrak{r} = (L \leftarrow (V_\text{i} \cup V_\text{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with basic events B and let $B' \supseteq B$. Let $L_{FDEP} = L_{SPARE} = R_{FDEP} = R_{SPARE} = \emptyset$ and $A = \{\{v_1, v_2\} \mid \text{ActivationConnection}(v_1, v_2) \subseteq \mathfrak{C}\} \subseteq \mathcal{P}(V_\text{i} \cup V_\text{o})\}$. Then $\mathfrak{r}$ is valid if*

$$\forall \pi \in B'^{\triangleright}\ \forall f \text{ oracle for } \mathfrak{r}$$
$$h_r(\text{Failed}_L[f](\pi) \cap V_\text{o}) = \text{Failed}_R[f](\pi) \cap h_r(V_\text{o})$$

and for all $\{v, v'\} \in \mathcal{P}_2(V_\text{i} \cup V_\text{o}) \setminus A$, it holds that

$$v \bowtie_L v' \iff h_r(v) \bowtie_R h_r(v'),$$

and for all $\{v, v'\} \in A$,

$$v \not\bowtie_L v' \implies h_r(v) \not\bowtie_R h_r(v').$$

Proof sketch of Theorem 5.16. We reuse the proof of Theorem 5.10. However, Item 11 on page 136 needs additional attention.

We cannot just show

$$\text{Active}_F(\pi) \cap (F_\text{SPARE} \cup F_\text{BE}) = \text{Active}_{F'}(\pi) \cap (F'_\text{SPARE} \cup F'_\text{BE})$$

as this no longer holds.

Instead, we show the stronger statement that

$$\begin{aligned}
&\forall \pi \in B'^{\rhd} \\
&\forall v \in \mathrm{Active}_F(\pi) \cap F_{\mathrm{SPARE}}\ v \in \mathrm{Active}_{F'}(\pi) \wedge \\
&\forall v \in \mathrm{Active}_F(\pi) \cap F_{\mathrm{BE}} \setminus \mathrm{Failed}(\pi)\ v \in \mathrm{Active}_{F'}(\pi) \vee \\
&\quad \forall \pi' \in B' \setminus \pi \cup \{v\}^{\rhd} \\
&\qquad \mathrm{top} \in \mathrm{Failed}(\pi \cdot \pi' \cdot v) \iff \mathrm{top} \in \mathrm{Failed}(\pi \cdot \pi') \wedge \\
&\qquad \mathrm{ClaimedBy}(\pi \cdot \pi' \cdot v) = \mathrm{ClaimedBy}(\pi \cdot \pi') \wedge \\
&\qquad \Delta(\pi \cdot \pi' \cdot v) = \Delta(\pi \cdot \pi')
\end{aligned}$$

That is, either the element is activated as before, or the failure of the basic event has no influence on the underlying Markov automaton.

The first case goes along the same lines as in Item 11 on page 136. We show

$$\forall v \in \mathsf{EMR}_F, \mathsf{EM}_{F,v} \cap (X \cup Y \cup Z) = \mathsf{EM}_{F',v} \cap (X \cup Y \cup Z).$$

We consider $(v, v') \in \mathsf{EMR}_F \times F_{\mathrm{SPARE}} \cup F_{\mathrm{BE}}$ such that $\mathrm{spmp}_F(v, v') \neq \emptyset$ and $\mathrm{spmp}_{F'}(v, v') = \emptyset$. Instead of being able to replace a removed path fragment by another fragment through the replaced subgraph, we may also replace the path fragment by some path in F, provided that for $v_{i'}$ and $v_{i''}$ as in Item 11 on page 136, we have $\mathrm{ActivationConnection}(v_{i'}, v_{i''}) \subseteq \mathfrak{C}$.

The proof that the conditions of the second case suffice goes along the lines of the proof of Theorem 5.7 on page 128, where we show that we can remove unconnected basic events. These conditions are indeed implied by the second condition from Definition 5.19. Any basic event which is not connected to a representant does not influence the failure of the top-level. Moreover, such basic event does not influence spare-gates (as it is not connected to a spare module representant) and no dependent events, as it is not connected to functional dependencies. □

5.3.2. Adding FDEPs

We now consider adding FDEPs to the rules, that is, we discuss the correctness of rules which add or remove FDEPs from the original DFT.

If we review the proof of Theorem 5.10, Item 8 on page 136 we see that we can allow the use of FDEPs if we prove that the set of dependent events after any event trace is unchanged. This happens, e.g., when the dependent event is required to have failed before the trigger can fail. Notice that also Item 1 on page 133 requires some attention, but in case the set of dependent events is unchanged, the argument is identical line by line.

Theorem 5.17. *Let $\mathfrak{r} = (L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with basic events B and let $B' \supseteq B$. Let $L_{SPARE} = R_{SPARE} = \emptyset$. Then $\mathfrak{r}$ is valid if*

$$\begin{aligned}
&\forall \pi \in B'^{\rhd}\ \forall f \text{ oracle for } \mathfrak{r} \\
&\quad h_r(\mathit{Failed}_L[f](\pi) \cap V_{\mathfrak{o}}) = \mathit{Failed}_R[f](\pi) \cap h_r(V_{\mathfrak{o}})\ \wedge\ \Delta_L[f](\pi) = \Delta_R[f](\pi)
\end{aligned}$$

and for all $\{v, v'\} \subseteq V_{\mathfrak{i}} \cup V_{\mathfrak{o}}$ it holds that

$$v \bowtie_L v' \iff h_r(v) \bowtie_R h_r(v').$$

Proof. We adapt the proof of Theorem 5.10. It is trivial that Item 8 on page 136 still holds. □

Often, the dependent events may change, but triggering the dependent even has no effect on the other outputs anymore. As dependent events are basic events and therefore usually part of the input interface (unless they're dummy events), we need to assure that they're not connected to any other parts of the DFT (which are not matched by the rule).

Definition 5.20. Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_{\mathfrak{i}} \cup V_{\mathfrak{o}}) \rightarrow R, \mathfrak{C})$ with $v \in V_{\mathfrak{i}}$. The set $\mathrm{NoOtherPreds}(v) \subseteq \mathrm{DFTs} \times (L \to \mathrm{NoOtherPreds}(v)_1)$ is defined as

$$\{(F, \zeta) \mid \exists v' \in \theta(\zeta(v)) v' \notin \zeta(L)\}$$

A rewrite rule $\mathfrak{r}$ possesses the *no-other-predecessor context restriction w.r.t.* v if $\text{NoOtherPreds}(v) \subseteq \mathfrak{C}$. ■

We observe that the difference in dependent events is surely a subset of the right-hand side of the rule. We cannot longer ensure isomorphism of the underlying Markov chain - we thus aim for weak equivalence.

Assumption 4. *We simplify the reasoning in the following a bit by assuming that for the proofs after this, only one element is a trigger and that this trigger is part of the input interface.*

Whenever the trigger fails, other dependent events might first be handled, but this is ultimately the same thing as that the event is the trigger fails later. Now for each dependent event subset of dependent events that have failed after the trigger on the host DFT (left-hand side in the rule), their should be a (possibly empty) subset of the dependent events of this trigger in the resulting DFT (right-hand side in the rule) such that the sets of failed elements with predecessors in the DFT. Regarding the rewrite rule, that is all elements in the output-interface and the elements in the input-interface which are not restricted by the no-other-predecessor context restriction. The formal version is given in the theorem below.

Theorem 5.18. *Let* $\mathfrak{r} = (L \leftarrow (V_\text{i} \cup V_\text{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with basic events* B *and let* $B' \supseteq B$. *Let* $L_{SPARE} = R_{SPARE} = \emptyset$ *and* $|\{\sigma(v)_1 \in V_L \mid v \in L_{FDEP}\}| = \{x\}$ *for some* x. *Let* $A = \{v \mid \mathit{NoOtherPreds}(v) \subseteq \mathfrak{C}\} \subseteq V_\text{i}\}$. *Then* $\mathfrak{r}$ *is valid if*

$$
\begin{aligned}
&\forall \pi \in B'^{\rhd}\ \forall f \text{ oracle for } \mathfrak{r} \\
&\quad h_r(\mathit{Failed}_L[f](\pi) \cap V_\text{o}) = \mathit{Failed}_R[f](\pi) \cap h_r(V_\text{o}) \wedge \\
&\quad x \in \mathit{Failed}_L[f](\pi) \implies \forall C \subseteq \Delta_L[f](\pi) \exists C' \subseteq \Delta_R[f](\pi) \\
&\qquad \forall \pi' \in C^{\rhd} \forall \pi'' \in C'^{\rhd} \\
&\qquad\quad h_r(\mathit{Failed}_L[f](\pi \cdot \pi') \cap V_\text{i} \setminus A \cup V_\text{o}) = \mathit{Failed}_R[f](\pi \cdot A') \cap h_r(V_\text{i} \setminus A \cup V_\text{o})
\end{aligned}
$$

and for all $\{v, v'\} \subseteq V_\text{i} \cup V_\text{o}$ *it holds that*

$$v \bowtie_L v' \iff h_r(v) \bowtie_R h_r(v').$$

We omit a full formal proof here. The proof involves showing that the set of output-elements which have failed coincide after each possible order of dependent events, at each point during the failure of the dependent events and under consideration of any further failures of input-elements due to dependent failures outside the rule. This is directly guaranteed by the precondition in the theorem. That this indeed suffices is a repetition of the arguments in the proof of Theorem 5.10, Items 4 and 5 on page 134.

In other cases, we want to eliminate a functional dependency by using failure propagation via an or-gate. This is only possible under certain semantic assumptions, which were features in Section 4.4 on page 102. There, also syntactic criteria which account for these assumptions were given. We translate them into context restrictions here.

Definition 5.21. Give a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\text{i} \cup V_\text{o}) \rightarrow R, \mathfrak{C})$ with $v \in L_\text{BE}$. The set $\text{Preferential}(v) \subseteq \text{DFTs} \times (L \rightarrow \text{Preferential}(v)_1)$ is defined as

$$
\begin{aligned}
\{(F, \zeta) \mid \quad & \exists v' \in F_\text{BE}\ \exists x \in V_F \exists p, p' \in \text{ifcp}(x).\ p_\downarrow = \zeta(v) \wedge p'_\downarrow = v' \wedge \\
& \zeta(v) \text{ and } v' \text{ are not mutual commutative}\}
\end{aligned}
$$

A rewrite rule $\mathfrak{r}$ possesses the *preferential context restriction w.r.t.* v if $\text{Preferential}(v) \subseteq \mathfrak{C}$. ■

Definition 5.22. Give a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\text{i} \cup V_\text{o}) \rightarrow R, \mathfrak{C})$ with $v \in V_\text{i}$ and $v' \in L_\text{BE}$. The set $\delta\text{-Independent}(v, v') \subseteq \text{DFTs} \times (L \rightarrow \delta\text{-Independent}(v, v')_1)$ is defined as

$$\{(F, \zeta) \mid \exists x \in F_\text{BE} \exists p \in \text{ifcp}(\zeta(v)).p_\downarrow = x \wedge x \text{ is ignorant about } \zeta(v')\}$$

A rewrite rule $\mathfrak{r}$ possesses the *delta-independent context restriction w.r.t.* v *and* v' if $\text{Preferential}(v') \subseteq \mathfrak{C}$ and $\delta\text{-Independent}(v, v') \subseteq \mathfrak{C}$. ■

Earlier, we only discussed the semantical context restrictions due to the removal of connections and thereby creating additional spare modules. When replacing functional dependencies by regular connections, that is, replacing failure forwarding by failure combination, we regularly add connections. As we want to focus ourselves here on the failure mechanism, we introduce a rather severe restriction.

Definition 5.23. Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x \in V_\mathfrak{i} \cup V_\mathfrak{o}$. The set TopConnected$(x) \subseteq$ DFTs $\times (L \rightarrow \text{TopConnected}(x)_1)$ is defined as

$$\{(F, \zeta) \mid \text{spmp}(\text{top}_F, \zeta(x)) = \emptyset\}.$$

A rewrite rule $\mathfrak{r}$ possesses the *top-connected context restriction w.r.t.* x if TopConnected$(x) \subseteq \mathfrak{C}$. ■

The restriction suffices to ensure that elements are activated from the start. If no connectiones are eliminated, then the condition implies that the activation propagation is untouched.

Remark 35. By definition, elements which are not connected to any module representant are never activated. Connecting them to the top-level module causes them to be activated instantly, therefore affecting their behaviour. A dormancy rate different from zero for elements not connected to a spare module representant is syntactic sugar. We can thus safely assume that all dormancy rates for such elements are equals 1. During rewriting, we would have to enforce this assumption. That results in changes in the component failures, which we deliberately prevent in the context of this thesis.

We further simplify the theorem by assuming that we only remove a single functional dependency, notice that this has not a major impact, as the removal of several functional dependencies can be easily split in several rule applications. The theorem looks as follows.

Theorem 5.19. *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with basic events* B *and let* $B' \supseteq B$. *Let* $L_{SPARE} = R_{SPARE} = R_{FDEP} = \emptyset$ *and* $|L_{FDEP}| = \{x\}$ *for some* x. *Let* $A = \{v \mid \textit{NoOtherPreds}(v) \subseteq \mathfrak{C}\} \subseteq V_\mathfrak{i}$, $A' = \{v \mid \textit{Preferential}(v) \subseteq \mathfrak{C}\} \subseteq V_\mathfrak{i}$ *and* $\hat{A} = \{(v, v') \mid \delta\textit{-Independent}(v, v') \subseteq \mathfrak{C}\} \subseteq V_\mathfrak{i} \times V_\mathfrak{i}$ *such that*

$$\sigma(x)_2 \in A \cap A' \cap \{y \mid (x, y) \in \hat{A}\}.$$

Then $\mathfrak{r}$ *is valid if*

$$\begin{aligned}
&\forall \pi \in B'^{\triangleright} \\
&\forall f \textit{ oracle for } \mathfrak{r}\ \forall f' \textit{ oracle for } \mathfrak{r} \textit{ with } f'(\pi) = f(\pi) \cup \{\sigma(x)_2\} \\
&\quad \sigma(x)_1 \notin f(\pi) \implies h_r(\textit{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \textit{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o}) \wedge \\
&\quad \sigma(x)_1 \in f(\pi) \implies h_r(\textit{Failed}_L[f'](\pi) \cap V_\mathfrak{o}) = \textit{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})
\end{aligned}$$

and for all $v \in V_\mathfrak{i} \cup V_\mathfrak{o}$ *it holds that*

$$\textit{TopConnected}(v) \subseteq \mathfrak{C}.$$

We omit a formal proof here. The significant difference to the earlier proof obligation occurs when $\sigma(x)_1 \in \text{Failed}_L[f](\pi)$ and $\sigma(x)_2 \notin \text{Failed}_L[f](\pi)$. In those cases, we are only interested in equivalence after the dependent event has been triggered.

5.4. DFT rewrite rules

In this section, we present a selection of rules to illustrate the usage of the framework as well as the variety of rules that are possible. We choose to display them graphically, as this eases understanding the given rules.

Notation We choose to present potentially infinite families of rules. To obtain a concrete rule from a family, concrete values for any variable occurring have to be selected, as well as instantiating all but input nodes' types. Both concretisation of variables and instantiating types should follow all given restrictions. Some input nodes are marked as optional, which means that the rule is correct

both with and without optional nodes. In case we want to obtain the rule without the optional node present, we also ignore the corresponding rule on the right-hand side.

The following list describes the conventions used to display the families.

- The subDFT L is depicted on the left-hand side, while the subDFT R is depicted on the right-hand side.
- We label all elements of the graph with a unique identifier, although these labels do not occur in the formal definition. These labels help us to clearly formulate our proofs and our homomorphisms.
- Elements without a type are depicted by an upwards-pointing triangle, as used in [VS02] to depict so-called transfer elements (elements which are shown in a separate drawing). Notice however that we also use this in, e.g., Rule 1, to depict a family of rewrite rules.
- As in Chapter 4 on page 71 we use dotted arrows to denote functional dependencies, pointing from the trigger to the dependent element.
- Input and output elements are given by stating their identifiers in L. We then map them to the identifiers in R to display the homomorphism from $V_\mathsf{i} \cup V_\mathsf{o}$ to R.
- Any restrictions on variable concretisation or type instantiation, as well as optional nodes (denoted opt.), are depicted with curly braces in the representation of L.

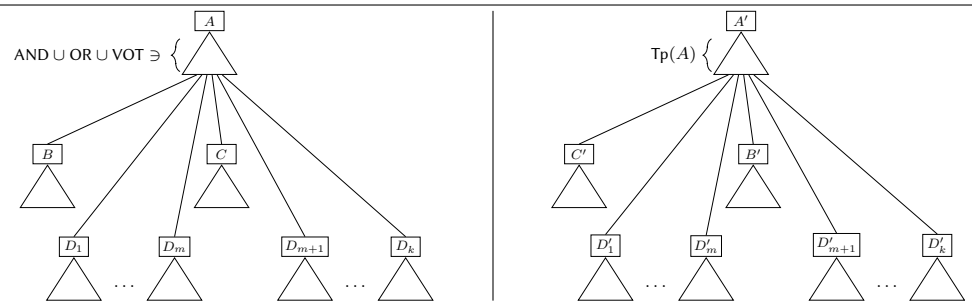
We choose to display just a selection of the corresponding correctness proofs, as they all follow the same scheme. We start with the rules regarding static elements and pand-gates. It is important to notice that the rules are applicable in general DFTs.

5.4.1. Static elements and the pand-gate

We start with some general rules and simplifications of chains of binary gates to n-ary gates. We then show the rules which originate from the lattice axioms for AND and OR. We continue with constant elements and voting gates and finish with different rules for PANDs.

Structural identities The first three rules describe general rules which follow directly from the characterisation of failed in Proposition 4.1. We have commutativity of static gates, as we do not consider ordering of the successors in the definitions.

Rewrite rule 1 Commutativity of static gates



Input: $\{D_i \mapsto D_i'\}_{i=1}^{k} \cup \{B \mapsto B', C \mapsto C'\}$

Output: $\{A \mapsto A'\}$

Context Restriction: none

Proposition 5.20. *Rewrite rule 1 is valid and symmetric.*

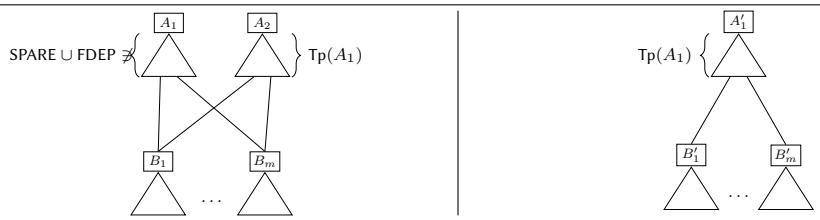
Proof. We show the proof for the case that $\mathrm{Tp}(A) = \mathrm{VOT}(k)$, this includes the proofs for AND and OR. We use the proof obligation from Theorem 5.10. We use the notation from there.

$$\begin{aligned} & A \in \mathrm{Failed}_L[f](\pi) && \iff \\ & |\{B, C, D_1, \ldots, D_k\} \cap \mathrm{Failed}_L[f](\pi)| \geq k && \iff \\ & |\{B', C', D'_1, \ldots, D'_k\} \cap \mathrm{Failed}_R[f](\pi)| \geq k && \iff \\ & A' \in \mathrm{Failed}_R[f](\pi) \end{aligned}$$

For the second part, it suffices that $\{X, Y\} \subset V_{\mathrm{i}} \cup V_{\mathrm{o}}$, $X \bowtie_L Y$ and $h_r(X) \bowtie_R h_r(Y)$. □

The next rule corresponds to what is commonly referred to as *functional congruence*. That is, the output of a (dynamic) function depends on its inputs. Put it differently, if all successors are equal, two elements with the same type surely fail simultaneously.

Rewrite rule 2 Gates with identical types and successors



Input: $\{B_i \mapsto B'_i\}_{i=1}^m$

Output: $\{A_1 \mapsto A'_1, A_2 \mapsto A'_1\}$

Context Restriction: none

Proposition 5.21. *Rewrite rule 2 is valid.*

The proof follows directly from the definition.

Gates with just a single child fail together with this child, so they can directly be eliminated.

Rewrite rule 3 Gate with only one successor

Input: $\{B \mapsto B'\}$

Output: $\{A \mapsto B'\}$

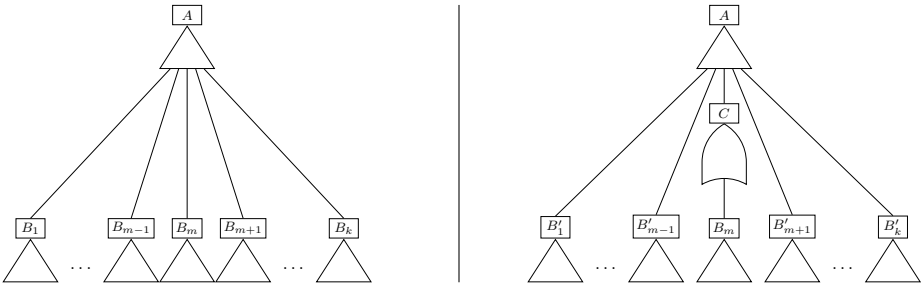
Context Restriction: none

Proposition 5.22. *Rewrite rule 3 is valid.*

The proof follows directly from the definition.

To keep other rules (especially Rewrite rule 24) more general, it is beneficial to also define a rule which replaces a successor with an or

Rewrite rule 4 Add an OR in between.



Input: $\{B_i \mapsto B'\}_{i=1}^{k}$

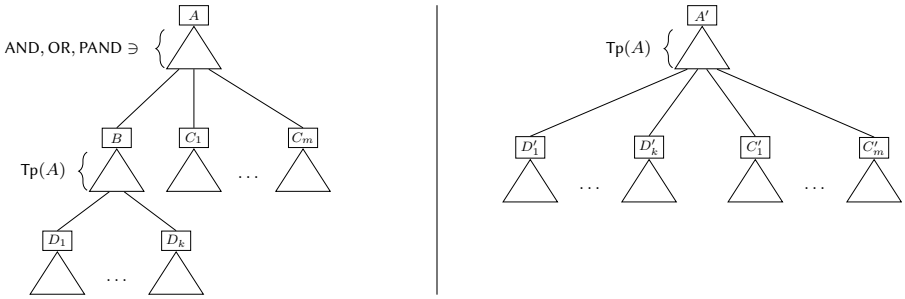
Output: $\{A \mapsto A'\}$

Context Restriction: none

Proposition 5.23. *Rewrite rule 4 is valid.*

As we support n-ary gates, an element with a first successor of the same type can be merged with that successor. For commutative gates, this would also work for arbitrary successors, but using Rewrite rule 1, we can reorder the successors before and after to apply the rule. For the pand-gate, it is not possible to apply this on arbitrary successors.

Rewrite rule 5 Left-flattening of and/or/pand gates



Input: $\{C_i \mapsto C_i'\}_{i=1}^{m} \cup \{D_i \mapsto D_i'\}_{i=1}^{k}$

Output: $\{A \mapsto A'\}$

Context Restriction: none

Proposition 5.24. *Rewrite rule 5 is valid and symmetric.*

Proof. We show the proof for PAND. The proofs for AND and OR are analogous (with a simpler structure as the order does not matter).

For the validity, we use the proof obligation from Theorem 5.10. The rule does not include any spare-gates or functional dependencies.

For the first part of the proof obligation, we have to show that $A \in \text{Failed}_L[f](\pi) \iff A' \in \text{Failed}_R[f](\pi)$.

We notice that the following holds for any $x \in \text{PAND}$:

$$x \in \text{Failed}(\pi) \implies \forall \pi' \in \text{pre}(\pi).\ \text{FB}_{\pi'}(x, \sigma(x)_{\downarrow}) \wedge \text{FB}_{\pi'}(\sigma(x)_{\downarrow}, x)$$

Then, we have

$$
\begin{aligned}
&A' \in \text{Failed}_F[f](\pi) && \Longrightarrow \\
&C'_1,\dots,C'_m,D'_1,\dots,D'_k \in \text{Failed}_R[f](\pi) \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C_i,C_{i+1}) \\
&\quad \wedge \text{FB}_\pi(D'_k,C'_1) \wedge \bigwedge_{i=1}^{k-1} \text{FB}_\pi(D'_i,D'_{i+1}) && \Longrightarrow \\
&C_1,\dots,C_m,D_1,\dots,D_k \in \text{Failed}_L[f](\pi) \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C_i,C_{i+1}) \\
&\quad \wedge \text{FB}_\pi(D_k,C_1) \wedge \bigwedge_{i=1}^{k-1} \text{FB}_\pi(D_i,D_{i+1}) && \Longrightarrow \\
&C_1,\dots,C_m,B \in \text{Failed}_L[f](\pi) \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C_i,C_{i+1}) \wedge \text{FB}_\pi(D_k,C_1) && \Longrightarrow \\
&C_1,\dots,C_m,B \in \text{Failed}_L[f](\pi) \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C_i,C_{i+1}) \wedge \text{FB}_\pi(B,C_1) && \Longrightarrow \\
&A \in \text{Failed}_L[f](\pi)
\end{aligned}
$$

and

$$
\begin{aligned}
&A \in \text{Failed}_L[f](\pi) && \Longrightarrow \\
&B,C_1,\dots,C_m \in \text{Failed}_L[f](\pi) \wedge \text{FB}_\pi(B,C_1) \wedge \bigwedge_{i=1}^{m} \text{FB}_\pi(C_i,C_{i+1}) && \Longrightarrow \\
&B,C_1,\dots,C_m \in \text{Failed}_L[f](\pi) \wedge \text{FB}_\pi(D_k,B) \wedge \text{FB}_\pi(B,C_1) \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C_i,C_{i+1}) && \Longrightarrow \\
&D_1,\dots D_K,C_1,\dots,C_m \in \text{Failed}_L[f](\pi) \wedge \text{FB}_\pi(D_k,C_1) \\
&\quad \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C_i,C_{i+1}) \wedge \bigwedge_{i=1}^{k-1} \text{FB}_\pi(D_i,D_{i+1}) && \Longrightarrow \\
&D'_1,\dots D'_K,C'_1,\dots,C'_m \in \text{Failed}_R[f](\pi) \wedge \text{FB}_\pi(D'_k,C'_1) \\
&\quad \wedge \bigwedge_{i=1}^{m-1} \text{FB}_\pi(C'_i,C'_{i+1}) \wedge \bigwedge_{i=1}^{k-1} \text{FB}_\pi(D'_i,D'_{i+1}) && \Longrightarrow \\
&A' \in \text{Failed}_R[f](\pi)
\end{aligned}
$$

For the second part, we simply observe that for all $\{X,Y\} \subset V_\text{i} \cup V_\text{o}$, $X \bowtie_L Y$ and $h_r(X) \bowtie_R h_r(Y)$. This directly implies the requested property.

For symmetry, we observe that h_r is injective and that we have a unrestricted context. By Proposition 5.11, the rule is symmetric. □

If an element does not have a predecessor, its failure is never propagated to any other elements. Therefore, it can be removed. The element may, however, connect two otherwise unconnected subgraphs, thereby constructing a single module. Thus, the rule can only be applied in restricted contexts (cf. Example 5.1 on page 113).

Rewrite rule 6 Remove elements without a predecessor.

Input: $\{B_i \mapsto B'_i\}_{i=1}^m$

Output: $\emptyset$

Context Restriction: $\{\text{ActivationConnection}(B_i, B_j) \mid 1 \leq i < j \leq m\}$.

Proposition 5.25. *Rewrite rule 6 is valid.*

Proof. We use the proof obligation from Theorem 5.16, as we have a context restriction. The failure part is trivially fulfilled, as there are no output elements. The second part of the proof is trivially fulfilled, as $\mathcal{P}(2)V_\text{i} \cup V_\text{o} \setminus \{B_i, B_j \mid 1 \leq i < j \leq m\} = \emptyset$. □

Especially due to rewriting, a gate may have multiple successors. For AND and ORs, we can safely remove one of them, we can safely assume that only one edge exists. For PAND gates, this is true whenever the two successors are neighbours, i.e. no other successor is ranked in between. We give a rule which eliminates the first successor if it is identical to the second one. Due to commutativity, this yields a general rule for AND and OR. For PAND gates, this also suffices when used in combination with Rewrite rule 20.

Rewrite rule 7 First two successors of AND/OR/PAND identical.

Input: $\{C_i \mapsto C'_i\}_{i=1}^m \cup \{B \mapsto B'\}$

Output: $\{A \mapsto A'\}$

Context Restriction: none

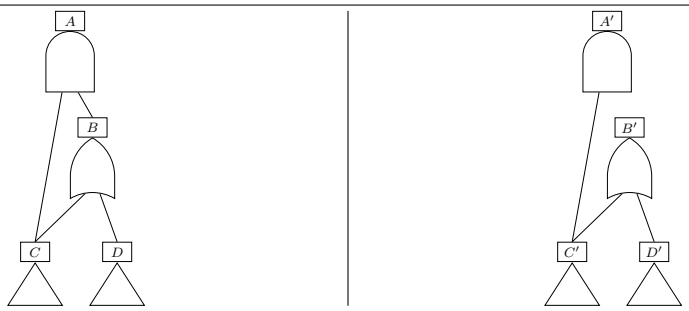
Proposition 5.26. *Rewrite rule 7 is valid and symmetric.*

The proof follows directly from the definition.

Rules originating from lattice axioms All lattice axioms (cf. Table 5.1 on page 114) can be translated into rules for DFTs. The commutativity is already handled in Rewrite rule 1. Associativity for n-ary gates is covered by the flattening rule (Rewrite rule 5).

We present the subsumption rule used in the framework. The here presented rules are slightly different from the rule used in Example 5.1 on page 113.

Rewrite rule 8 Subsumption of or-gate by an and-gate.



Input: $\{C \to C', D \to D'\}$

Output: $\{A \to A', B \to B'\}$

Context Restriction: none

We define the rule as presented above instead of the rule as in earlier examples as the here presented rule is more general. It allows B to have other predecessors. In case it does have other predecessors, only the connection is removed, thereby simplifying A, but B is not removed. In case no other predecessors exist, B can afterwards be eliminated by Rewrite rule 6.

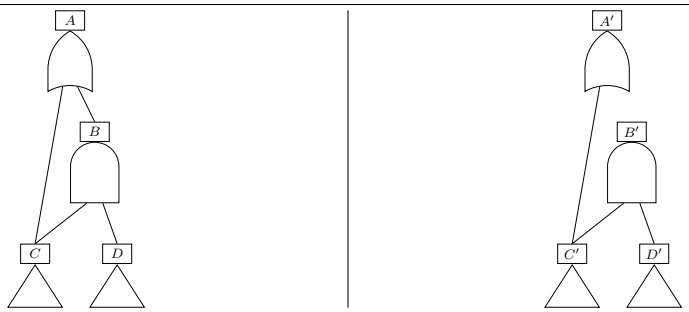
Proposition 5.27. *Rewrite rule 8 is valid and symmetric.*

The rule is presented only for the binary case, as we do repeatedly afterwards, in order to keep the representation and the proof obligation as simple as possible. Notice that this is not a restriction. Using the flattening, we can apply the rules also on n-ary graphs. We illustrate this interplay of rewrite rules in the following example.

Example 5.7. We consider an extension of the DFT from Example 5.1 on page 113 in Figure 5.13a. We want to apply the subsumption rule. As the gates are not binary, as required by the rule, we want to make them binary. In order to do this, we want to apply deflattening. As we only defined left-(de)flattening, we use commutativity (Rewrite rule 1, on B) to reorder the successors (Figure 5.13b). We can now apply deflattening (Rewrite rule 5, on A and B both times in reverse direction), and we obtain (Figure 5.13d). Reordering the successors of B (Rewrite rule 1) yields Figure 5.13c). Here, we can apply the rewrite rule for subsumption. We obtain Figure 5.13e. We restore the successors of the and-gate by flattening (Rewrite rule 5, on Z and B), which yields Figure 5.13f. We can remove the or-gate B as it has no predecessors and the context restriction is met (Rewrite rule 6, no spare gate) and subsequently, we can remove the or-gate Z. This yields Figure 5.13g. We can now remove the unconnected basic events D and X, following Theorem 5.7. We arrive at the final result depicted in Figure 5.13h. ▲

The subsumption rule for or-gates over and-gates is analogous to the rule presented above.

Rewrite rule 9 Subsumption of or-gate by an and-gate.

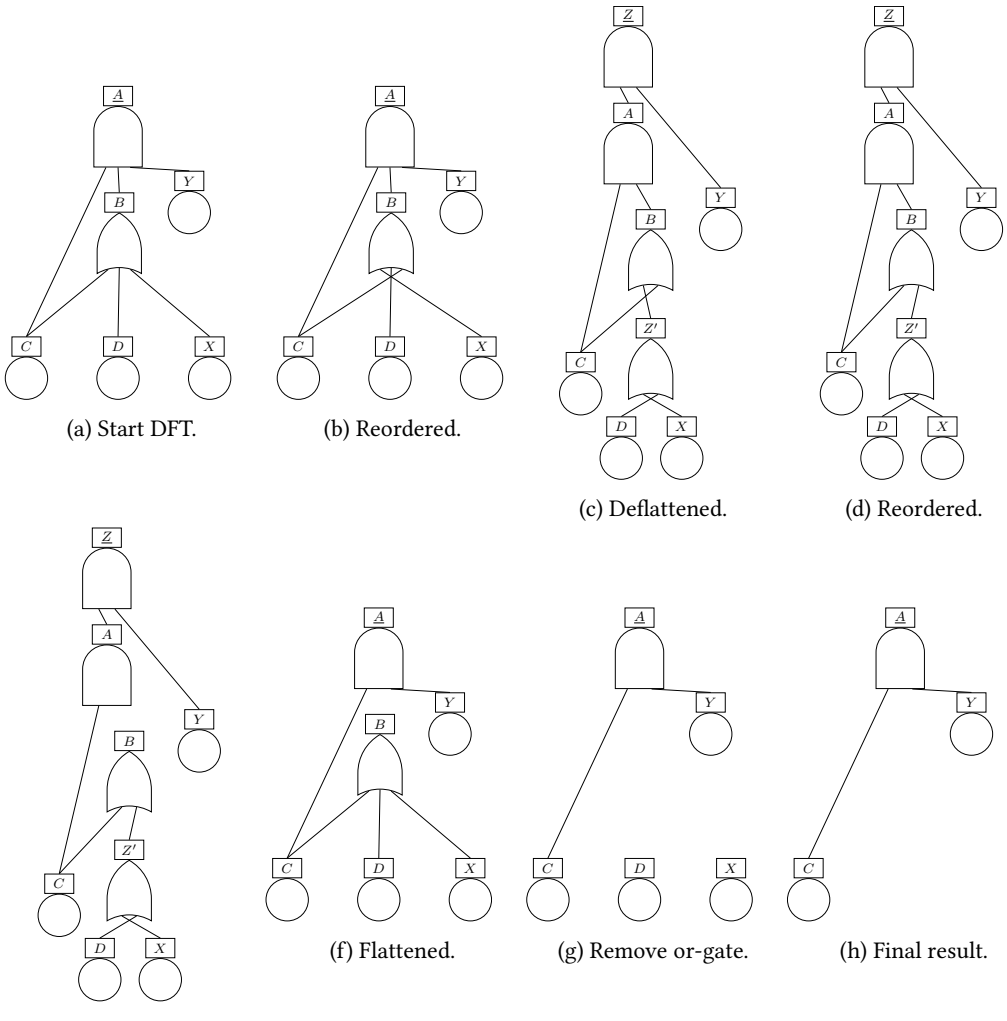


Input: $\{C \to C', D \to D'\}$

Output: $\{A \to A', B \to B'\}$

Context Restriction: none

Proposition 5.28. *Rewrite rule 9 is valid.*



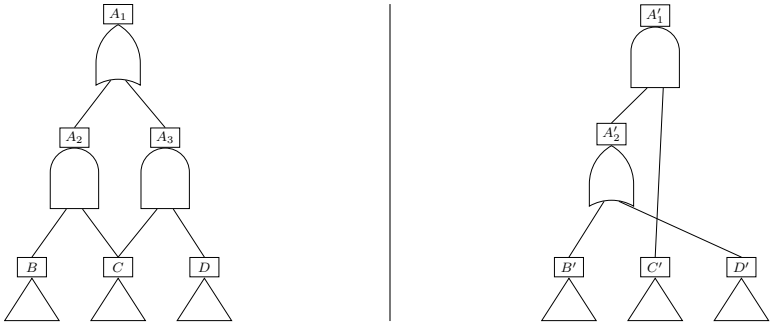
(a) Start DFT. (b) Reordered. (c) Deflattened. (d) Reordered.

(e) Subsumption applied. (f) Flattened. (g) Remove or-gate. (h) Final result.

Figure 5.13.: Applying subsumption in a larger DFT as explained in Example 5.7 on page 147

The distribution rewrite rule directly originates from the lattice axioms. As for subsumption, we present the rule for binary gates. Using (de)flattening, we can do this for n-ary gates.

Rewrite rule 10 Distribution of or-gates over and-gates.



Input: $\{B \mapsto B', C \mapsto C', D \mapsto D'\}$

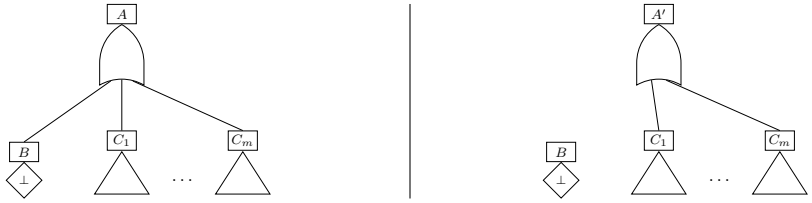
Output: $\{A_1 \mapsto A'_1\}$

Context Restriction: none

Proposition 5.29. *Rewrite rule 10 is valid and symmetric.*

Based on the lattice axioms, we expect a number of rules to eliminate constant elements. In fact, we have that either a constant successor can be removed, as it does not affect the gate (e.g. Rewrite rule 11) or the constant element is propagated upwards, as it determines the outcome of its predecessor gate (e.g. Rewrite rule 12). As we remove connections in the rule, we have to ensure that the activation context remains intact.

Rewrite rule 11 Or-gate with an infallible successor.



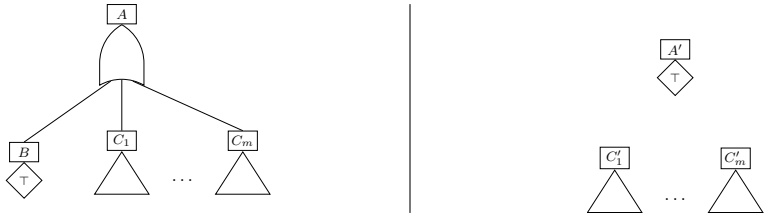
Input: $\{C_i \mapsto C'_i\}_{i=1}^{m}$

Output: $\{A \mapsto A', B \mapsto B'\}$

Context Restriction: $\{\text{ActivationConnection}(B, A)\}$

Proposition 5.30. *Rewrite rule 11 is valid.*

Rewrite rule 12 Or-gate with a constant fault as successor.



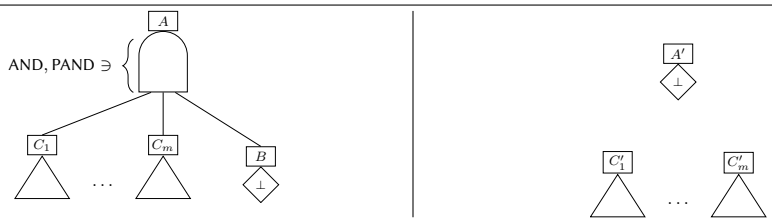
Input: $\{C_i \mapsto C'_i\}_{i=1}^{m}$

Output: $\{A \mapsto A', B \mapsto A'\}$

Context Restriction: $\{\text{ActivationConnection}(A, C_i) \mid 1 \leq i \leq m\}$

Proposition 5.31. *Rewrite rule 12 is valid.*

Rewrite rule 13 And gate with an infallible successor



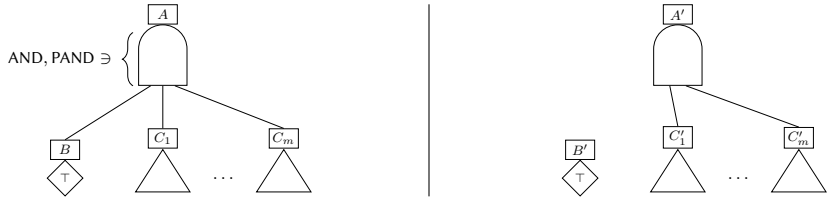
Input: $\{C_i \mapsto C'_i\}_{i=1}^m$

Output: $\{A \mapsto A', B \mapsto A'\}$

Context Restriction: $\{\text{ActivationConnection}(A, C_i) \mid 1 \leq i \leq m\}$

Proposition 5.32. *Rewrite rule 13 is valid.*

Rewrite rule 14 And-gate with a constant fault as successor.



Input: $\{C_i \mapsto C'_i\}_{i=1}^m$

Output: $\{A \mapsto A', B \mapsto B'\}$

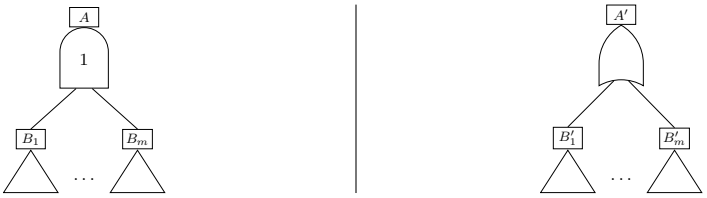
Context Restriction: $\{\text{ActivationConnection}((A, B))\}$

Proposition 5.33. *Rewrite rule 14 is valid.*

Voting gate The AND and OR are both special voting gates, so we expect a rule which originates from this. Furthermore, the voting gate can be modelled by a number of ORs and ANDs. We thus expect a rule for the conversion to a representation of voting gates to AND and ORs.

We start with the representation of OR and AND as voting gate.

Rewrite rule 15 Voting with threshold 1 is an or-gate.



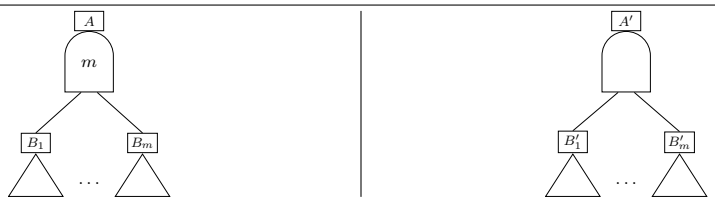
Input: $\{B_i \mapsto B'_i\}_{i=1}^m$

Output: $\{A \mapsto A'\}$

Context Restriction: none

Proposition 5.34. *Rewrite rule 15 is valid and symmetric.*

Rewrite rule 16 Voting with m successors and threshold m is an and-gate.



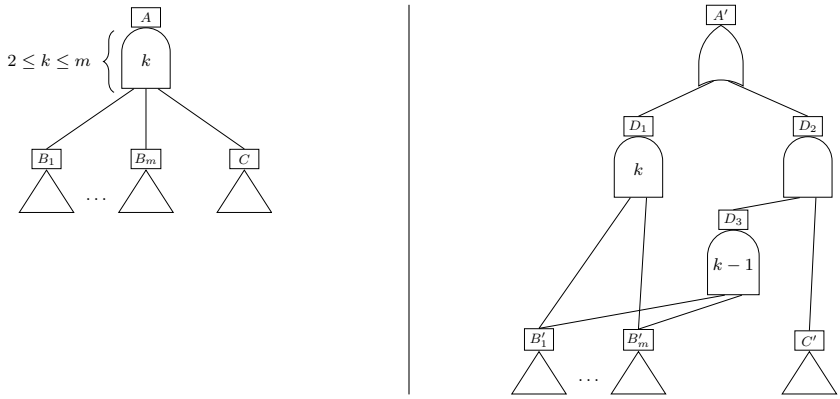
Input: $\{B_i \mapsto B_i'\}_{i=1}^{m}$
Output: $\{A \mapsto A'\}$
Context Restriction: none

Proposition 5.35. *Rewrite rule 16 is valid and symmetric.*

For the expansion of voting gates, we could construct a disjunctive normal form, where we would encode all possible combinations that k out of m successors have failed. However, such a rule would yield an extreme blow-up. Instead, we use a rule based on the Shannon expansion. Shannon expansion is based on the following proposition in Boolean algebra, where $f(x_1, \ldots, x_n)$ denotes a Boolean function over the variables $x_1 \ldots x_n$:

$$f(x, y_1, \ldots, y_n) = x \wedge f(1, y_1, \ldots, y_n) \vee \neg x \wedge f(0, y_1, \ldots, y_n).$$

Rewrite rule 17 Shannon expansion



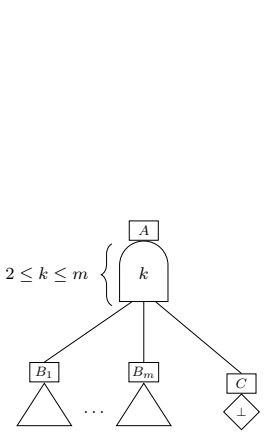
Input: $\{B_i \mapsto B_i'\}_{i=1}^{m} \cup \{C \mapsto C'\}$
Output: $\{A \mapsto A'\}$
Context Restriction: none

In the given rule, the successor C corresponds to the first variable in the Boolean function. Either it fails, then the output of the rule fails already with $k-1$ out of the m other successors failed. Moreover, the output element fails certainly when k out of the remaining m successors fail - independent of the failure of C.

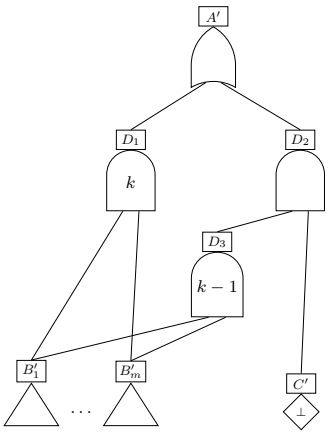
We can combine these rules to eliminate constant elements as successors, which we illustrate in Figure 5.14 on page 152.

Conflicting and combined sequences We focus ourselves on PANDs. As PANDs are AND gates which a restriction of the order of sequences, we expect some rules which allow us to combine PANDs.

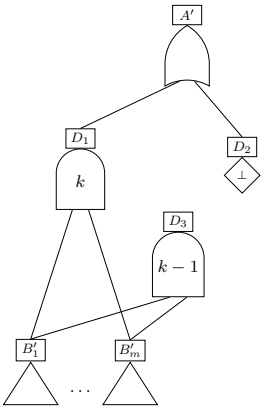
An and-gate is a pand-gate without any ordering requirements. One could express this as all possible sequences connected by an or-gate. This is exactly what is done in the next rule, were we express the requirement that A and B both occur by requiring that either first A and then B or first B and then A occurs. From left-to-right, the rule allows us to subsequently combine pand-gates with other pand-gates. From right-to-left, we get an obvious simplification.



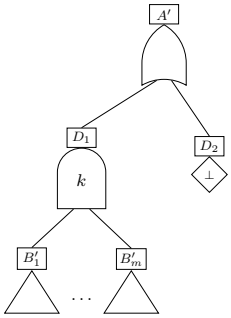
(a) Initial subDFT.



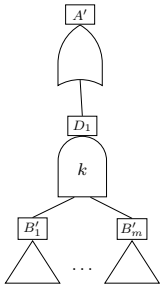
(b) Apply Shannon expansion (Rewrite rule 17)



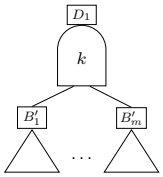
(c) Eliminate AND with a fail-safe element (Rewrite rule 13)



(d) Remove unconnected basic event according to Rewrite rule 6



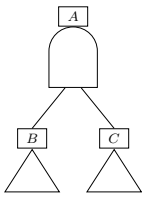
(e) Eliminate fail-safe element from OR (Rewrite rule 11)

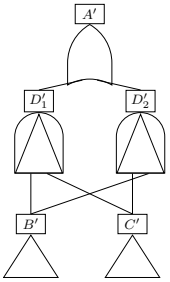


(f) Eliminate single child element (Rewrite rule 3)

Figure 5.14.: Steps for rewriting VOT(k) with a fail-safe element

Rewrite rule 18 And-gate using or- and pand-gates



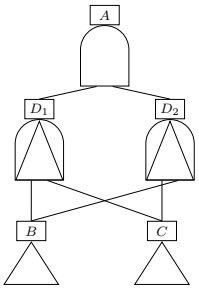


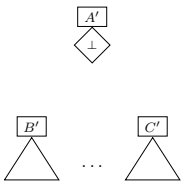
Input: $\{B \mapsto B', C \mapsto C'\}$
Output: $\{A \mapsto A'\}$
Context Restriction: none

Another particular instance is an AND gate which has two pand-gates as successors, where the order requirements of the PANDs are in mutual conflict. Under the assumption that the children never fail simultaneously, it can never be true that both PANDs fail. therefore, the and-gate cannot fail, as also discussed in Section 3.3.1 on page 34.

Notice that to prevent the simultaneous failure, we also need the event-independency for either B or C. With commutativity, it suffices to only present a rule with B event-independent.

Rewrite rule 19 Conflicting PANDs with independent children.





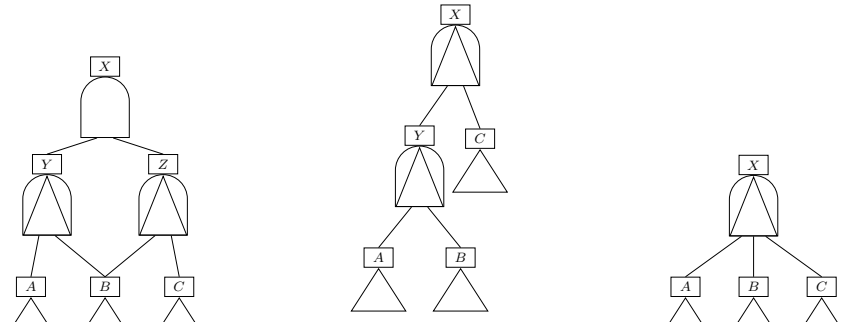
Input: $\{B \to B', C \to C'\}$
Output: $\{A \to A'\}$
Context Restriction: $\{\text{IndependentInputs}(B, C), \text{EventDependentFailure}(B)\}$
$\{\text{ActivationConnection}(A,B), \text{ActivationConnection}(A,C)\}$

Proposition 5.36. *Rewrite rule 19 is valid.*

Notice that we did not formally introduce any proof obligation for a combination of context restrictions. As the effects are orthogonal, the combination is straightforward.

PANDs with various gates as successors We consider simplification of pand-gates with a gate as a successor.

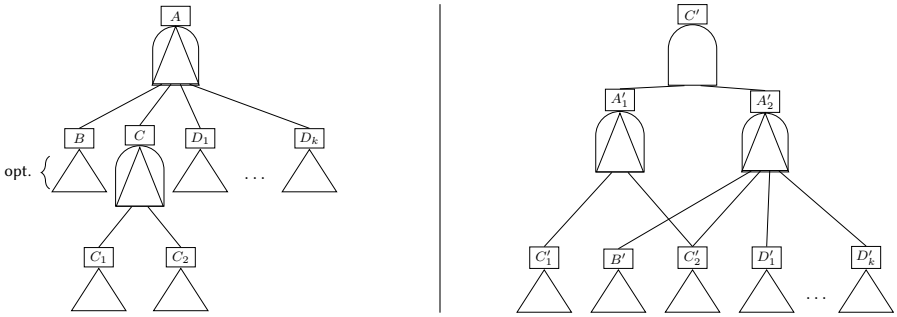
For the static gates, we can merge the gates with any successors of the same type. This is independent of their position, as the gates are commutative. For pand-gates, this does not hold. However, pand-gates with a pand-gate as successor can always be rewritten. We present a rule for pand-gates whose second successor is a pand-gate. This rule can be generalised to arbitrary positions of the successor by using the left-flattening (reverse) of the pand-gate to group all successors before the second pand-gate into a single new pand-gate.



(a) Start DFT. (b) Applying Rewrite rule 20 from right to left. (c) After flattening.

Figure 5.15.: Rewriting a chain of PANDs.

Rewrite rule 20 Pand-gate with a pand-gate as successor



Input: $\{B \mapsto B', C_1 \mapsto C'_1, C_2 \mapsto C'_2\} \cup \{D_i \mapsto D'_i\}_{i=1}^{k}$

Output: $\{A \mapsto C'\}$

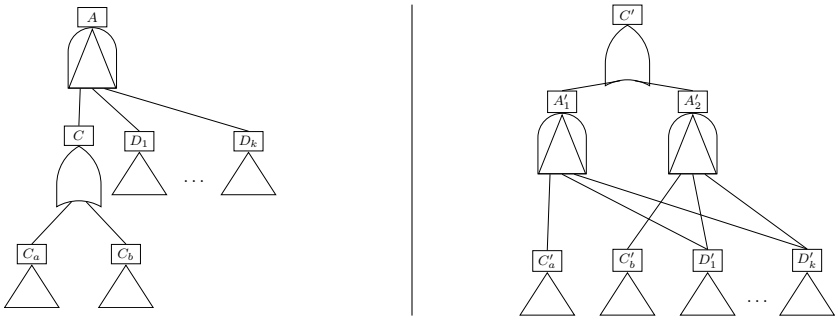
Context Restriction: none

Proposition 5.37. *Rewrite rule 20 is valid and symmetric.*

Notice that the rule from right-to-left has great practical relevance. It allows, together with the deflattening, to rewrite *pand-chains*, as found in, e.g. the sensor-filter case study (Figure 3.41 on page 62). We illustrate this in Figure 5.15.

For an or-gate as a non-first successor, we cannot find a general rule which removes the or-gates from the subtree of a pand-gate. This follows directly from Proposition 5.2, and was also illustrated in Section 3.3.4.1 on page 40. We can, however, lift an or-gate which is the first successor of a pand-gate.

Rewrite rule 21 Pand-gate with an or-gate as first successor



Input: $\{B_1 \mapsto B'_1, C_2 \mapsto C'_2\} \cup \{D_i \mapsto D'_i\}_{i=1}^{k}$

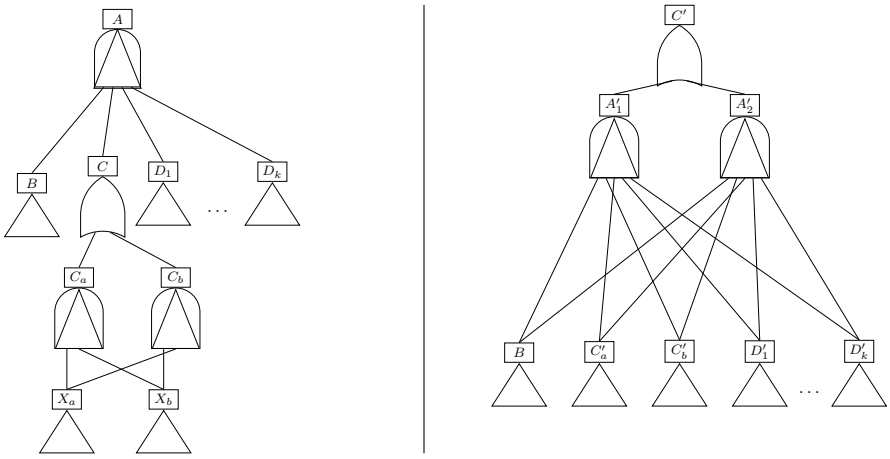
Output: $\{A \mapsto C'\}$

Context Restriction: none

Proposition 5.38. *Rewrite rule 21 is valid and symmetric.*

Notice that for special cases, there are useful rules to eliminate ORs which are non-first successors of PANDs. As an example, we give the following rule, which also allows rewriting and-gates below an pand-gate.

Rewrite rule 22 Pand-gate with an or-gate as successor (special case)



Input: $\{B \mapsto B', C_a \mapsto C'_a, C_b \mapsto C'_b\} \cup \{D_i \mapsto D'_i\}_{i=1}^{k}$

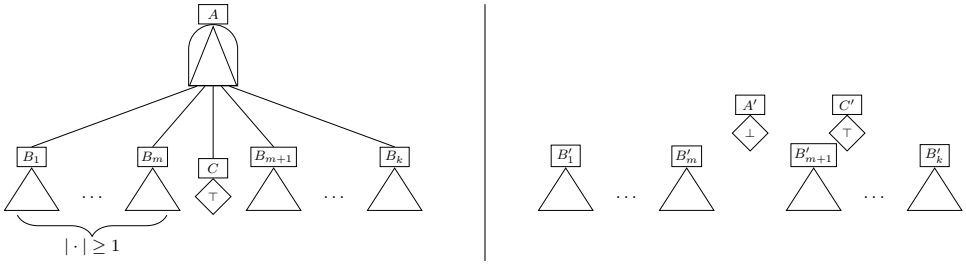
Output: $\{A \mapsto C'\}$

Context Restriction: none

Proposition 5.39. *Rewrite rule 22 is valid and symmetric.*

We already introduced the elimination of constant elements as first (last) successor of a PAND in Rewrite rule 14 and Rewrite rule 13. We need another rule for non-first successors encoding a given failure, as such an element means that the ordering-condition of the pand is violated if the previous successors have not failed at initialization. We ensure this via the event dependent context.

Rewrite rule 23 Pand gate with a constant fault as non-first successor



Input: $\{B_i \mapsto B'_i\}_{i=1}^{k}$

Output: $\{A \mapsto A', C \mapsto C'\}$

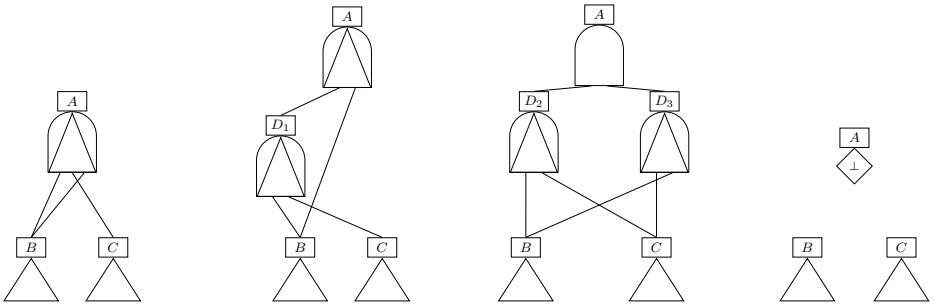
Context Restriction:
$\{\text{ActivationConnection}(X, B_i) \mid 1 \leq i \leq k \wedge X \in \{A, C\}\} \cup$
$\{\text{ActivationConnection}(B_i, B_j) \mid 1 \leq i < j \leq k\} \cup$
$\{\text{ActivationConnection}(A, C)\} \cup$
$\{\text{EventDependentFailure}(B_i) \mid 1 \leq i \leq k\}$

Proposition 5.40. *Rewrite rule 22 is valid.*

We can rewrite many other combinations of PANDs by the combination of the given rules. We give example of another combination of rewrite rules in Figure 5.16 on page 156

5.4.2. Rewrite rules with functional dependencies

In the following section, we present a selection of rewrite rules for functional dependencies. Rewriting functional dependencies is usually heavily context-sensitive, both from a syntactic as from a



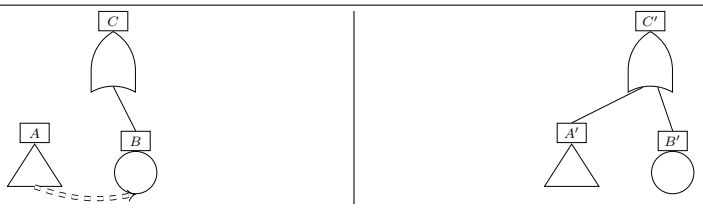
(a) Initial subDFT. (b) After application of flattening, in inverse direction (Rewrite rule 5). (c) Using Rewrite rule 20 (d) Assuming A, B independent, apply Rewrite rule 19.

Figure 5.16.: Steps for rewriting PAND with a duplicate child

semantical point of view.

We discussed earlier that in the context of static fault trees, functional dependencies are syntactic sugar. We thus want a rewrite rule which indeed eliminates functional dependencies. A single rewrite rule is sufficient for this. It is the analogous rewrite rule to the method described by Merle *et al.* in [MRL10]. It rewrites an functional-dependency into a an or-gate. The rule is neither component- nor hierarchy conservative, so context restrictions due to well-formedness criteria apply. Due to the simple structure we impose on input- and output- interface, the rule is slightly more verbose than one might expect, by the added or-gate which has but one successor.

Rewrite rule 24 Eliminating FDEPs by the introduction of a or-gate



Input: $\{A \mapsto A', B \mapsto B'\}$

Output: $\{C \mapsto C'\}$

Context Restriction: $\{\text{Preferential}(B),$
$\delta\text{-Independent}(A, B),$
$\text{NoOtherPreds}(B),$
$\text{TopConnected}(A), \text{TopConnected}(C)\}$

Proposition 5.41. *Rewrite rule 24 is valid.*

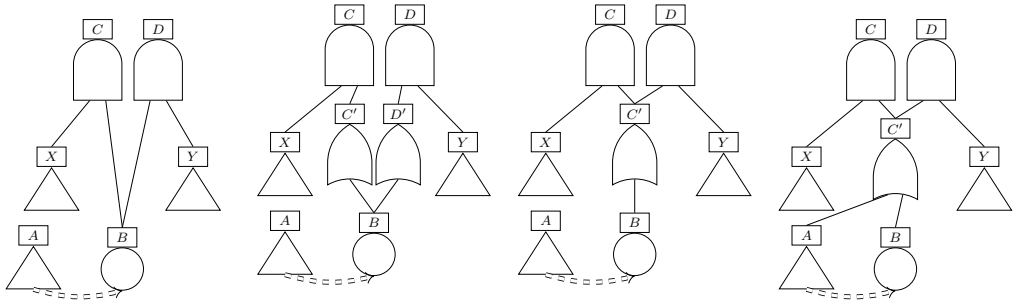
Proof sketch. We proof this by using Theorem 5.19.

Let f be an arbitrary oracle. We only discuss a subset of the oracles here, which is easily generalised to a complete proof. We therefore assume $\pi \in B'^{\triangleright} \setminus \{\varepsilon\}$ and $\pi' \in \text{pre}(\pi) \cup \{\pi\} \setminus \{\varepsilon\}$ arbitrary sequences such that $f(\pi') \neq f(\pi'_{|-1}, f(\pi) \neq f(\pi_{|-1})$ and $\forall \hat{\pi}\pi \in \text{pre}(\hat{\pi}) f(\hat{\pi}) = f(\pi)$.

The first scenario is: $f(\varepsilon) = \emptyset, f(\pi') = \{A\}, f(\pi) = \{A, B\}$. We have to show that

- $C \in \text{Failed}_L[f](\varepsilon) \iff C' \in \text{Failed}_R[f](\varepsilon)$.
  We have $f(\varepsilon) = \emptyset$ and thus $C \notin \text{Failed}_L[f](\varepsilon)$. Furthermore, $h_r(f(\varepsilon)) = \emptyset$, and thus $C' \notin \text{Failed}_R[f](\varepsilon)$.
- $C \in \text{Failed}_L[f'](\pi') \iff C' \in \text{Failed}_R[f'](\pi')$.
  It holds that $f'(\pi') = \{A, B\}$ and thus $C \in \text{Failed}_L[f'](\pi')$. Likewise, we have that $h_r(f'(\pi')) = \{A', B'\}$ and thus $C' \in \text{Failed}_R[f'](\pi')$.
- $C \in \text{Failed}_L[f'](\pi) \iff C' \in \text{Failed}_R[f'](\pi)$.
  As above, both $C \in \text{Failed}_L[f'](\pi)$ and $C' \in \text{Failed}_R[f'](\pi)$.

The second scenario, $f(\varepsilon) = \emptyset$ and $f(\pi') = f(\pi) = \{A, B\}$ is analogous to the third case above.



(a) Start DFT. (b) Add or-gates with single successor. (c) Merge or-gates with equal successors. (d) Rewrite FDEP.

Figure 5.17.: Circumventing the NoOtherPreds($\{B\}$) context restriction

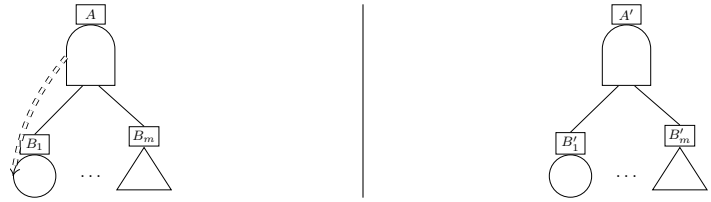
.

The third and last scenario we consider is $f(\varepsilon) = \{B\}$ and $f(\pi') = f(\pi) = \{A, B\}$. We show that $C \in \text{Failed}_L[f](\varepsilon) \iff C' \in \text{Failed}_R[f](\varepsilon)$. The rest follows by the coherency of failing. We have $f(\varepsilon) = \{B\}$ and thus $C \in \text{Failed}_L[f](\varepsilon)$, and analogously $C' \in \text{Failed}_R[f](\varepsilon)$. □

The restriction of no other predecessors ensures that all triggered failures are covered by the or-gate. This is not a restriction in the original sense, i.e. on each DFT on which the rule could be applied without the restriction, the rule can be applied by first modifying the DFT and then applying this rule. We illustrate this in Figure 5.17 on page 157.

We present some other rules for eliminating dispensable functional dependencies. Notice that in the remainder of this section, we do not aim for completeness, as the number of rules becomes too large. The presented rules serve as an overview of ideas for further rules which are not necessarily captured yet by a more general approach, as done for Rewrite rule 24.

Rewrite rule 25 Superfluous FDEPs from AND to a successor.



Input: $\{B_i \mapsto B'_i\}_{i=1}^m$

Output: $\{A \mapsto A'\}$

Context Restriction: none

Proposition 5.42. *Rewrite rule 25 is valid.*

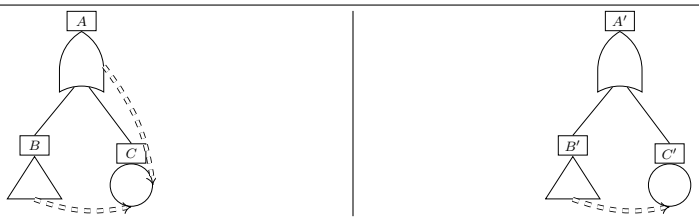
Proof. We use the proof obligation from Theorem 5.17. Let f be an oracle and π be an event trace.

$$A \in \text{Failed}_L[f](\pi) \iff \{B_1, \dots B_m\} \subseteq \text{Failed}_L[f](\pi) \iff$$
$$\{B'_1, \dots, B'_m\} \subseteq \text{Failed}_R[f](\pi) \iff A' \in \text{Failed}_R[f](\pi)$$

Furthermore, $\Delta_{\text{Lab}}(\pi) = \emptyset = \Delta_R(\pi)$ (the trigger A of the FDEP on the lhs fails only if the dependent event has failed before). Furthermore, we have that $A \bowtie_L B \bowtie_R C$ and $A' \bowtie_R B' \bowtie_R C'$. □

We notice that functional dependencies are anonymous, and we cannot distinguish multiple functional dependencies with the same dependent event. The elimination of the following functional dependency is therefore another example where the set of dependent events is untouched.

Rewrite rule 26 Superfluous FDEP from OR to successor.



Input: $\{B \mapsto B', C \mapsto C'\}$

Output: $\{A \mapsto A'\}$

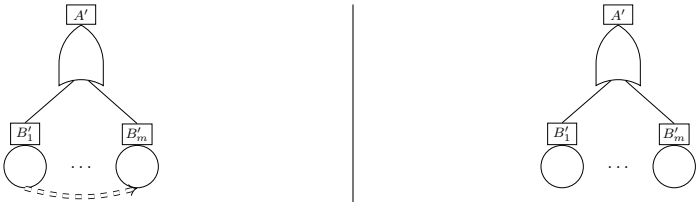
Context Restriction: none

Proposition 5.43. *Rewrite rule 26 is valid.*

Some functional dependencies are dispensable when their predecessors overlap, we give two common examples. Notice that, in general, the nodes could have multiple overlapping predecessors, which is not directly supported by these rules, but can be mimicked as in Figure 5.17. As the functional dependency potentially causes the failure of (input) interface elements, another proof obligation is required for the correctness proofs of the next few rules.

The next rule might seem as being easily simulated with the help of Rewrite rules 3 and 24. However, the restrictions from Rewrite rule 24 are strong and might prevent the rules from being applied in the scenario below.

Rewrite rule 27 Removing FDEP between successors of an OR.



Input: $\{B_i \mapsto B_i'\}_{i=1}^{m}$

Output: $\{A \mapsto A'\}$

Context Restriction: NoOtherPreds($\{B_m\}$)

Proposition 5.44. *Rewrite rule 27 is valid.*

Proof. We use the proof obligation from Theorem 5.18. Let f be an oracle and π be an event trace.

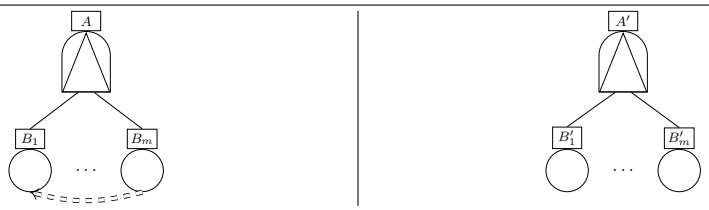
$$A \in \text{Failed}_L[f](\pi) \iff \{B_1, \dots B_m\} \cap \text{Failed}_L[f](\pi) \neq \emptyset \iff$$
$$\{B_1', \dots, B_m'\} \cap \text{Failed}_R[f](\pi) \neq \emptyset \iff A' \in \text{Failed}_R[f](\pi)$$

Furthermore, $B_1 \in \text{Failed}(\pi) \implies \Delta_L[f](\pi) \in \{\emptyset, \{B_m\}\}$ and $\Delta_R[f](\pi) = \emptyset$.

- For $\Delta_L[f](\pi \cdot \varepsilon) = \emptyset$ we we first notice that $B_i \in \text{Failed}_L[f](\pi \cdot \varepsilon) \implies A \in \text{Failed}_L[f](\pi \cdot \varepsilon)$ and furthermore that $B_i \in \text{Failed}_L[f](\pi \cdot \varepsilon) \implies B_i' \in \text{Failed}_R[f](\pi \cdot \varepsilon) \implies A' \in \text{Failed}_R[f](\pi \cdot \varepsilon)$. Furthermore, we have that for all $1 \leq i < m$, $B_i \in \text{Failed}_L[f](\pi \cdot \varepsilon) \iff B_i \in \text{Failed}_R[f](\pi \cdot \varepsilon)$.
- For $\Delta_L[f](\pi) = \{B_m\}$, the obligatons for $\pi \cdot \varepsilon$ are analogous to the case above. With coherency, we get $A \in \text{Failed}_L[f](\pi \cdot B_m)$. By the structure of L, it holds that for $1 \leq i < m$, $B_i \in \text{Failed}_L[f](\pi \cdot B_m) \iff B_i \in \text{Failed}_L[f](\pi)$ and thus $B_i \in \text{Failed}_L[f](\pi \cdot B_m) \iff B_i \in \text{Failed}_R[f](\pi \cdot \varepsilon)$. □

The following rule is another example of a functional dependency without effect. When the trigger fails, either the dependent event has already failed, which makes the FDEP superfluous. Otherwise, the PAND is rendered infallible. The subsequent failure of the dependent event does not change this. The failure of the FDEP has no other effect.

Rewrite rule 28 Removing FDEP between successor and a previous successor of an PAND.

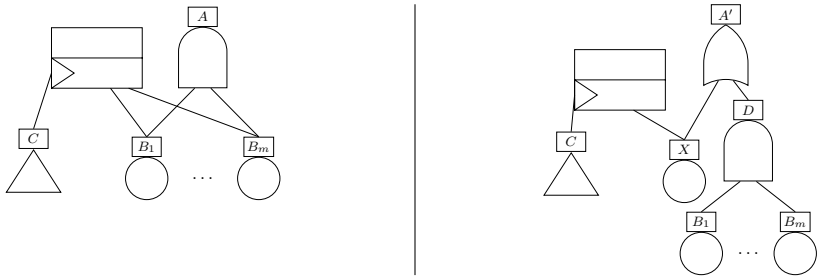


Input: $\{B_i \mapsto B_i'\}_{i=1}^m$
Output: $\{A \mapsto A'\}$
Context Restriction: NoOtherPreds($\{B_1\}$)

Proposition 5.45. *Rewrite rule 28 is valid.*

Sometimes, we can redirect the functional dependencies by the introduction of a dummy event. These dummy events can often be eliminated later.

Rewrite rule 29 Simplifying FDEPs in context of an AND.



Input: $\{B_i \mapsto B_i'\}_{i=1}^m \cup \{C \mapsto C'\}$
Output: $\{A \mapsto A'\}$
Context Restriction: NoOtherPreds($\{B_i\}_{i=1}^m$)

Proposition 5.46. *Rewrite rule 29 is valid.*

We stop here with the presentation of rules. We notice that we did not include any rules to eliminate or create spare gates, as, due to the activation propagation and claiming behaviour, such rules are far from trivial and their correctness seems to require a tremendous effort. While most rules for static gates are straightforward, the context restrictions due to the activation context require great care. The internal state of PANDs already yields trouble for finding a general normal form, as was illustrated by the example of a por-gate representation. However, especially functional dependencies can occur in many different contexts. The rules presented here merely give an idea of elimination in case the ordering is unaffected or the dependent events have no effect if the trigger has already failed. We excluded all kind of combinations of FDEPs here. Furthermore, we notice that functional dependencies have a major influence on keeping modules independent.

6. Experiments

This chapter presents the set-up for the experiments we use to assess the practical relevance of the rewriting procedure. In Section 6.1 on page 161, we give an overview over the implementation of the rewrite rules in Groove. In Section 6.2 on page 163, we briefly present the tool-chain around Groove, used to transform back and forth from DFTs to their graph representation. Section 6.3 on page 165 compares the performance of DFTCalc on several benchmark sets. To this end, it contains a description of the used instances, an detailed analysis of the performance of DFTCalc on these sets and a comparison of the rewritten instances with the original instances.

6.1. Groove grammar for DFTs

We implement the graph rewriting of DFTs within Groove, described in Section 2.3.2 on page 19. We stress that we cannot one-to-one transform our rewrite rule families to Groove:

- Context restrictions have to be handled differently.
- Families cannot be coded as presented in Section 5.4 on page 141, neither is symmetry supported.
- Groove does not use DPO rewriting, therefore, we have to encode the interface differently.

Notice that Groove allows the use of quantifiers, which notably simplifies the encoding of some rules. We first present the way we encode some of the rules. We then consider in which combination the rules should be applied in order to yield a simplification.

Remark 36. The full set of rewrite rules, together with the control program, can be found at `http://moves.rwth-aachen.de/ft-diet/`.

6.1.1. Concrete grammar

Like the graph representation of DFTs, we use some auxiliary nodes in addition to nodes encoding elements. As Groove has native supports for types, we use these types to encode the element type. We use the type-graph as depicted in Figure 6.1. Here, elements are a hierarchy. That is, an element in a DFT is, from the perspective of Groove, either a constant element or a basic event, or a static gate, or a dynamic gate. Static gates, such as AND and OR, have elements as successors. Notice that these successors are unordered. In contrast, dynamic gates, such as here the PAND and SPARE (WSp for warm spare), have auxiliary ordering-nodes, which each have a single element as a child. The ordering nodes under a dynamic gate are connected via a chain of next-edges, which encode the ordering. The top-level element is a special element, which has exactly one child. Furthermore, functional dependencies are encoded by specially labelled edges. In Figure 6.2 we depict a DFT and its Groove representation. We see that the AND, OR and PAND gate, as well as the three basic elements directly correspond to nodes in the Grooverepresentation. An additional node with one child marks the top-level and the children of the dynamic gate are ordered with the help of auxiliary nodes.

We present four rewrite rules here. We stress that due to the encoding, we do not need rules such as commutativity.

The encoding of Rewrite rule 8 in Groove, depicted in Figure 6.3 seems straightforward. However, the represented rule is more general than the original counterpart, as both the AND and the OR are allowed to have additional children, which we explicitly prohibit for the original DFT rewrite rules. The encoding of Rewrite rule 5, depicted for or-gates in Figure 6.4, shows a way to explicitely encode a family, as the rule accepts an arbitrary number of successors for both or-gates. The encoding of Rewrite rule 15 (Figure 6.5) shows the explicit encoding of an interface. Here, all

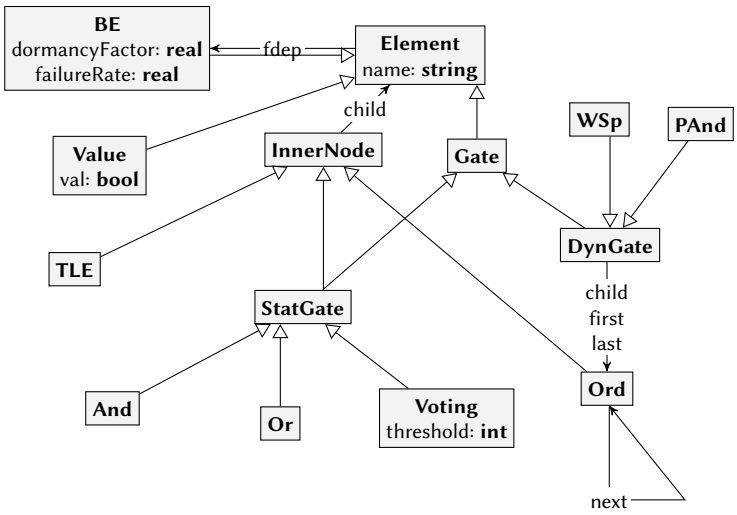
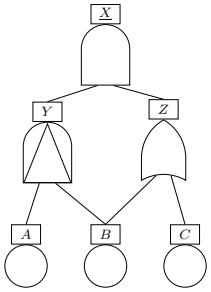
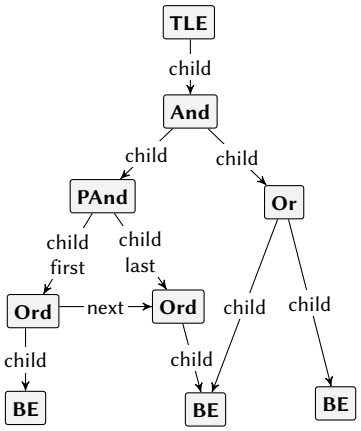


Figure 6.1.: Groove type-graph for DFTs.



(a) Example DFT.



(b) Representation in Groove.

Figure 6.2.: Representing a DFT in Groove.

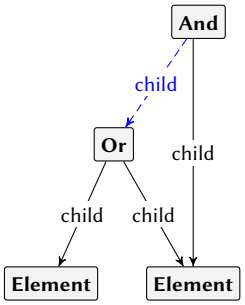


Figure 6.3.: Subsumption of an OR.

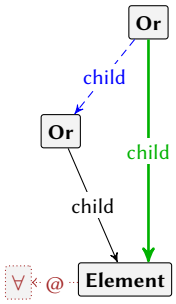


Figure 6.4.: Flattening an OR.

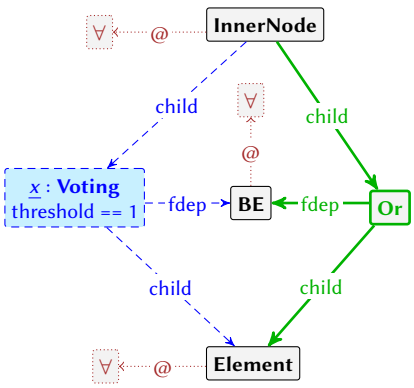


Figure 6.5.: VOT(1) equals OR.

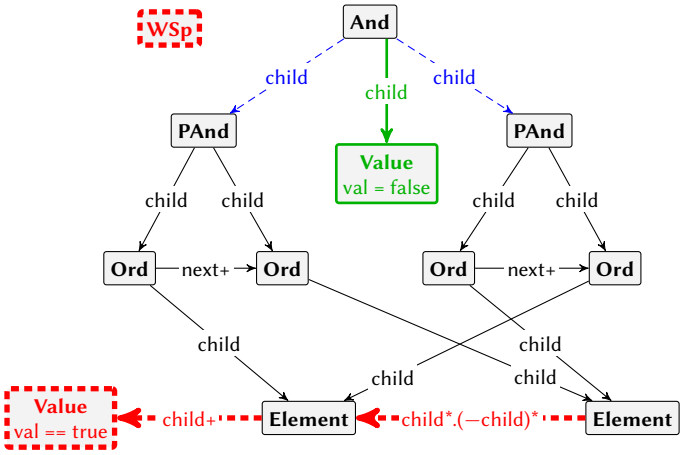


Figure 6.6.: Groove rule for conflicting PANDs.

edges to predecessors of the voting gate (including all functional dependencies) are replaced with edges from the predecessor to the new or-gate.

For Rewrite rule 19 a significant extension in the representation is required. The Groove representation is encoded in Figure 6.6 on page 163. We see that the independence criterion is easily expressed by stating that there is not a path from the one to the other element in the output interface such that it first goes down an failure-cause path and then (in reverse direction) goes up the failure-cause path again. Furthermore, the event-dependency is encoded by requiring one of the elements to not have an given-failure element in the successor-closure. The activation-connection is a much more difficult criterion to encode. We further restrict the rule here to be only applied if there is no spare-gate in the DFT, which certainly is a stronger restriction than required. To handle such larger context-restrictions, it is advisable to use a three-step procedure, in which a first rule adds information about contexts to the graph, the second performs the actual rewriting while considering the context restrictions now explicitly present in the DFT, and the third cleans up artefacts of the added information.

6.1.2. Control

Until now, we did not really discuss what rules, and what order of rules, leads to a simplification of the DFT. For this thesis, we restrict ourselves to a fairly simple approach, in which a priority is assigned to each rule. In each step, the rule with the highest priority is selected, until we find a state in which no further rule is applicable anymore.

The priority we assign to the rules is driven by a partitioning of the existing rules into *cleaning*, *reduction* and *transformation* rules. Some rules throw away dispensable elements without affecting the remainder of the DFT (*cleaning rules*). Such rules should always be applied directly, as they also prune the search space for other rule. Moreover, a large group of rules, such as rules for subsumption, seemingly reduce the size of the DFT directly. When no cleaning rules are available, we apply one of these *reduction rules*. When no other rules are applicable, we try rules such as the Shannon expansion for voting gates (*transformation rules*), as they might lead to subsequent reductions. Notice that the rules as used in the control program are confluent, that is, the control program always terminates at some point.

During the development, we tried out some more exotic constructs of DFTs, which indeed required more complex control programs to yield performance boosts. Moreover, we discovered that this control program is also not optimal for the given benchmark set, that is, by manually applying rules, we were able improve the performance on some instances.

6.2. Implementation details

We embedded Groove in a tool chain to allow simple usage of the rewrite framework[1].

[1] Available at http://moves.rwth-aachen.de/ft-diet/.

File format Common tools described in literature use the Galileo format (GalileoDFT, *.dft) as file format for the specification of DFTs. We slightly extended upon this format to support the full syntax as described in Chapter 4 on page 71. Please notice that our extension is fully backward compatible.

GroovyDFT As discussed in Section 6.1, we use Groove for the actual rewriting of DFTs. Groove uses the GXL [HSSW06] format as a disk format for storing graphs. Moreover, as discussed in Section 6.1 on page 161, the graph representation of DFTs used within Groove (GrooveDFT) is slightly different from the DFT graph. As the goal of utilising Groove is to support automatic rewriting of DFTs, we also need a tool for converting GalileoDFT (as *.dft) to GrooveDFT (as *.gst) and vice versa. We provide this functionality in the tool GroovyDFT.

GroovyDFT is implemented using Scala, and can thus be run from the Java runtime, like Groove. It consists of a total of about 800 lines of code, excluding comments. GroovyDFT provides data structures for GalileoDFT and GrooveDFT representation, as well as transformation code and marshalling/unmarshalling of these data structures. Moreover, it is able to export different metrics of a given fault tree, the most simple being the number of elements. We depict the components of GroovyDFT in Figure 6.7.

We briefly discuss two features. First of all, GalileoDFT does not support encoding evidence (or any constant elements at all). DFTCalc uses a separate flag to list basic events which are constantly failed, and infallible elements are simply encoded as dummy events. GroovyDFT therefore also accepts a separate list of evidence. Such basic elements are then replaced by constant failure elements before transformation to a graph-like structure. Equivalently, constant elements occurring in the GrooveDFT are substituted by basic events and a separate list with evidence is exported. Second, while producing GalileoDFT files, we normalise the file. That is, we list gates before basic events and all gates occur topologically sorted. For static gates, we order all successors based on, amongst others, the number of basic events in the subtree.

Remark 37. Another functionality is an elementary export of Tikz, which was used to draw the DFTs in this thesis.

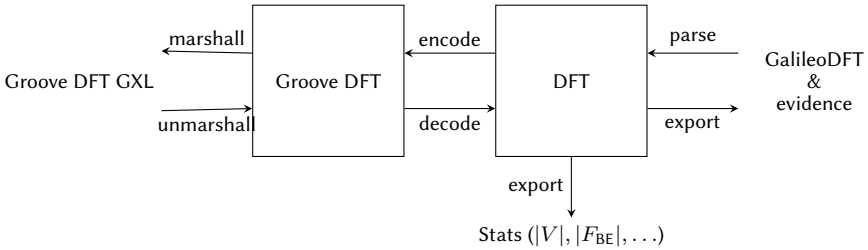


Figure 6.7.: Architecture of GroovyDFT.

Putting it together We wrapped the necessary tools in a little tool chain called aDFTpub[1], written in Python 3.

The tool takes an input DFT in GalileoDFT format and a measure (e.g. $\min |V|$) and produces an output DFT in GalileoDFT format. We depict the process in Figure 6.8. The tool chain transforms the input to GrooveDFT format by calling GroovyDFT, additionally storing some statistics exported by the tool. After the transformation by Groove with a given (set of) control programs, all the resulting graphs are transformed back to GalileoDFT format, again collecting statistics as outputted by GroovyDFT. Then, all DFTs are compared against the given entry for the statistics, and the DFT scoring best on the given measure is selected and exported by aDFTpub, while the others are discarded. Please notice that in some cases, the tool chain may export the input DFT as output DFT, if none of the rewritten DFTs score better on the selected measure.

Remark 38. In the experiments discussed next, we always use the (only) DFT which is returned by Groove.

[1] Another DFT by PUsh Button

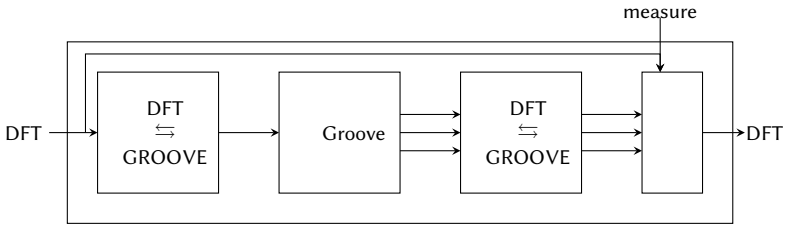


Figure 6.8.: Architecture of aDFTpub.

6.3. Experimental results

In this section, we report on the effects of rewriting DFTs on the performance of DFTCalc. To be more precise, we selected different benchmark sets from the literature. The performance of DFTCalc was tested by letting DFTCalc assess the reliability for a single mission time. This performance was compared against the performance of the tool chain, rewriting the DFT and only then feed the resulting DFT to DFTCalc to assess the reliability. Before we present the results, we first introduce the selected benchmark sets in greater detail[1] and discuss the performance measures we used.

6.3.1. Benchmarks for rewriting

We selected some of the case studies from Section 3.4 on page 55 as benchmarks. We wanted to include some of the most-used benchmarks, as well as the industrial ones. Furthermore, the DFTs should follow the standards in the hierarchical way they represent the system behaviour. As we did not consider por-gates for rewriting, those case studies were not suitable. As we want to show the scalability of the approach, it was important that we would have sufficiently large models.

We choose to use HECS, RC, MCS and SF as scalable benchmarks for which we write generators. The choice is mainly inspired by the availability of multiple sizes of these benchmarks and the straightforward scaling. Furthermore, we include HCAS (commonly used) and SAP (industrial + evidence required), as well as some confidential fault trees from Movares (partially described in [GKSL+14]).

HECS For this benchmark describing a computer system no scaled variants have been presented in literature. We consider a system which consists of multiple (m) (identical) computer systems of which k are required to be operational in order for the system to be operational. As the memory-interface structure with two interfaces is a bit complex, we also define systems with only one memory interface (all memory components in a computer fail if the interface fails) and a system where we do not consider the memory interface. We use $i = 0 \ldots 2$ to denote the number of interfaces. Furthermore, we consider two types of power supply (ps). The DFTs either do not consider the failure of a power supply (np). Alternatively, we consider DFTs where all computers have a power supply (part of the interface) which is functionally dependent on the power grid (up). We denote an instance of the benchmark as `h-`m`-`k`-`i`-ps`.

Please notice that we never produce gates with just one successor. Moreover, voting gates which correspond to an AND or an OR are replaced by those gates.

RC For this benchmark of a railroad crossing, we are not aware of any scaled variants. We consider a railroad crossing consisting not of just one barrier and one set of sensors. Instead, we generalise the concept such that a railroad crossing fails whenever any of the sensor-sets fail, or any of the barriers fail, or the controller fails. We then consider scalable versions with b identical barriers and s sets of sensors (each with their own cable which can cause a disconnect). Moreover, we consider a variant for the controller (ct). Either the controller failure is represented by a single basic event (sc) or by a computer described as in HECS (`h-1-1-2-np`, hc). We denote an instance of the benchmark as `rc-`s`-`b`-ct`.

[1] All used instances — including information about the used failure rates — are available from http://moves.rwth-aachen.de/ft-diet/

We notice that the generated benchmarks are, like the original benchmark, not compatible with the syntactic restrictions we use. As the interpretation of the semantics for this case in unambiguous, we ignore this issue. Again, no gates with only one successor is used.

MCS The MCS benchmark set has been scaled in literature to change the number of modules to four. We extend this scaling to higher numbers and in other dimensions.

As with the HECS computer system, we consider a farm of m computing systems where k are required to be operational. Each computing system contains i modules. A spare memory module is shared amongst two computing modules. If i is uneven, then one memory module is shared among three computing modules. All computing modules within a system share the same power supply (cps), which is either represented via an or-gate (x) or via FDEPs (f). In this thesis, we only included the variant with the or-gates. The power supply (pss) is either a single basic event (sp) or consists of two redundant power supplies (two basic events as successors of an AND, dp). We denote an instance of the benchmark set with `cm-`m`-`k`-`i`-pss-cps`.

As for HECS, gates with a single successor are eliminated and voting gates corresponding to AND or OR are converted

SF The sensor-filter is scalable benchmark for the Compass project. We could only recover single instances, and constructed a new benchmark set from it. We consider the combination of m filters. Any filter failing causes the top-level to fail. In each filter, we scale the number of steps k through the degraded states that have to be taken in order to cause a failure (in Figure 3.41 on page 62, we see three basic events with two PANDs). We denote an instance of the benchmark with `sf-`m`-`k.

HCAS We use the benchmark HCAS and CAS as presented in the literature. We included some varieties in the number of pumps.

SAP We use the benchmark as presented in the literature. We consider the four combinations of give failures and fail-safe elements for (BE1) and (BE3).

Movares The benchmark set contains a set of confidential fault trees representing pars of a railroad network. Some information is available from [GKSL+14]. The instances generally contain a large number of or-gates which are subject to be flattened. Dynamic elements are found in the lower levels of the fault tree to describe spare elements for specific components.

6.3.2. Performance of DFTCalc

Before we discuss the actual effect of the rewriting on the performance of DFTCalc, we discuss some observations we made in both during preparation and executing the benchmarks. These observations are important to understand the effect of rewriting on the performance of DFTCalc.

Reviewing the DFTCalc algorithm Let us first briefly discuss the internal algorithm used by DFTCalc. We recall from Section 3.5.5 on page 69 that each element is represented by an IMC. The underlying model used for the stochastic analysis is a larger IMC which is obtained by the parallel composition of the of IMCs corresponding to the single elements. As this parallel composition grows exponentially, a method called *smart reduction* is applied. That is,

- after (roughly) each composition, reduction techniques (e.g. constructing the bisimulation quotient) are applied.
- the order in which we apply the parallel composition is guided by a heuristic which aims to keep the bisimulation quotient small.

This yields a series of models. In Table 6.1 we give some key information about such a series of models.

We see that the model sizes alternate between growing and shrinking in most step. Growing models correspond to the application of composition, while shrinking models are obtained by reducing them. If the model size shrinks after a step where composition is expected, then two different IMCs were composed, yielding a separate IMC which might be smaller than the earlier

# states	# transitions	memory size (KB)
24	55	3.0
5	8	3.0
19	41	3.0
4	6	3.0
752	9360	20.5
440	4992	13.4
23	66	3.1
16	44	3.2
73	257	3.4
10	26	3.1
12	24	3.0
6	10	3.0
60	267	3.5
12	49	3.2
47	161	3.4
17	57	3.3
7058	77334	151.0
3724	38115	85.0
4924	43268	83.1
1476	12324	29.4
2952	18826	37.3
1727	11042	27.8
2354	12711	26.4
942	5131	14.7
1704	5750	12.7
79	283	4.0

Table 6.1.: The size of the intermediate models.

constructed IMC. The shown list shares some characteristics with many other computations of DFTCalc. First of all, the peak size is often not at the very end, but closer to the middle. Furthermore, this peak size is a true peak, i.e. most models are significantly smaller.

The last model obtained is used for the calculation of the requested stochastic measures on DFTs by executing either IMCA[1] or MRMC[2].

The vast majority of the computation resources are spent in the model construction, rather than in one of the model checkers, as the resulting models are mostly comparably small. The proposed rewriting mainly targets improving the overhead of the construction of the model. However, we execute all benchmarks by giving a DFT and a mission time, and requiring the reliability of the described system. This involves a single call to MRMC. We think that this is a better comparison, as it shows the timing differences for the most common use case. In some use cases, the user might be interested in a series of mission times, which yields a larger share of the computation time for MRMC.

Interpretation of the tables We present all our benchmarks in a table with the same columns, to ease the navigation through the table. We explain the entries of the table using two entries from the HECS benchmark, given in Table 6.2 on page 168.

Beginning in the second row, we see HECS indicating the name of the benchmark collection. Below, we see two rows starting with `h-1-1-1-np` and `h-1-1-1-up`, respectively. These are the names of instances in the benchmark collection. The entries in those rows reflect the performance on the baseline instances. The rows under the baseline start with slightly indented entries, printed in italic font. These rows reflect the names of the variants we created for the instance. Here, *simplified* reflects the name of the rewrite control program we used to rewrite the original

[1] Interactive Markov Chain Analyzer[GHHK+13]
[2] Markov Reward Model Checker[KZHH+11]

instances. The next two columns give the number of elements in the DFT ($|V|$) and the number of basic elements $|F_{\text{BE}}|$. The next three columns represent the intermediate model with the highest number of states (given in $\max_i |S_i|$) and their memory footprint (Mem, in kilobytes). In ($\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$), we show for each of the variants the relative memory footprint with respect to the baseline. The next two entries represent the final model size. We give the number of states ($|S_n|$) and the relative size of the variants with respect to the baseline ($\frac{|S_n|_{\text{bs}}}{|S_n|_{\text{rw}}}$). The last three columns show the time consumption. We give the time consumed by DFTCalc to construct the model and to calculate the reliability for a fixed mission time (t_D, in seconds) and the time consumption of Groove to rewrite the DFT from the baseline into the given variant (t_G, in seconds). The last column ($\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$) represents the speed-up relative to the baseline. It corresponds to the fraction of the time required to compute reliability on the baseline divided by the total time required to rewrite the baseline to the variant and compute the reliability on the variant ($t_D + t_G$). The peak memory consumption during the model creation is closely related to the memory footprint of the largest intermediate model.

All experiments are executed with a 2 hour timeout and a memory limit of 8 gigabytes. Computations which were cancelled due to these limits are marked with TO or MO, respectively. Details of the platform we used for the benchmarks can be found in Appendix B on page 197.

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert\mathbf{S_n}\rvert_{\text{bs}}}{\lvert\mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
HECS										
h-1-1-1-np	21	13	7058	151		79		75		
simplified	13	9	123	37.2	2.5	12	6.6	22	9	2.5
h-1-1-1-up	24	15	831	20		18		84		
simplified	13	9	123	4	5	5	3.6	5	9	5.8

Table 6.2.: Fraction of benchmark results.

Before we start with the actually used benchmarks, we show some variables for the DFTCalc heuristic which we have to be aware of, as well as the overhead of many small reductions, illustrated by comparing the seqand-gate with the spare-gate, and a case of a time/memory trade-off in the illustrated by finding the best representation of large static gates.

Encoding of the input In Section 6.2 on page 164, we described that during the export of the files, we also reorder the list of elements. While this is not definable on the syntax of DFTs, the heuristic within DFTCalc depends on it, as shown by shuffling the entries of the rewritten version of (see Table 6.3).

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert\mathbf{S_n}\rvert_{\text{bs}}}{\lvert\mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
cm-1-1-3-sp	29	15	1042	13.2		79		61		
shuffled	29	15	2050	24	0.6	12	15.2	61		1

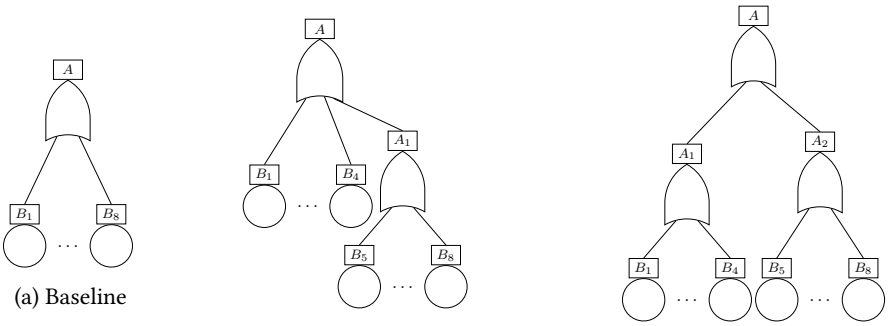
Table 6.3.: The effect of reordering the list of elements.

As we aim not to measure these effects but just the rewriting of the fault tree, we normalise all encodings. Please notice that ordering obtained by the normalisation is not necessarily unique, i.e. equivalent trees may be encoded (slightly) differently, however, we did not experience any performance differences between the normalised files. Furthermore, the goal of this normalisation is not to improve the performance, in fact, the ordering is not necessarily optimal. Instead, by enforcing the use of the same encoding for the baseline as well as for the simplified DFTs, we prevent structural bad or good encodings after generation or rewriting. With the large number of encodings, we achieve a high confidence that the trend shown in this benchmarks is indeed due to the effect of rewriting the DFT and not due to the difference in structure of the generated and rewritten DFTs.

Order and static gates We recall that the static gates are all commutative. Nevertheless, the order of the successors does influence the performance of DFTCalc, as shown in Table 6.4.

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert \mathbf{S_n}\rvert_{\text{bs}}}{\lvert \mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
h-1-1-1-np	21	13	7058	151		79		75		
shuffled	21	13	10124	217	0.7	81	1	79		1

Table 6.4.: The effect of the order of successors of a static gate.



(a) Baseline (b) Variant B (c) Variant C

Figure 6.9.: Three variants of an or-gate with eight BEs.

We recall that the successors of a static gate are unordered in Groove. Therefore, the rewriting process does not allow to optimise the order. To clean the experiments from the effect of this ordering, we normalise the order for all DFTs (also the baseline) using GroovyDFT.

For both the order of the encoding and the order of static gate successors, we experienced major impact on the peak sizes, but not on the time consumption.

Seqand-gates and the overhead of many small compositions In Section 4.2.9 on page 98 we discussed that seqand-gates are special cases of spare-gates. In fact, the IMC representation of seqand-gates can be simpler as it is not necessary to communicate the availability between different spare-gates. Therefore, we expect better improvements if all cold spare-gates with unshared spare components are rewritten to seqand-gates.

However, we see a significant performance drop when using seqand-gates (Table 6.5). To understand why, we regard the intermediate model sizes given in Table 6.6. The DFT contains 4 binary spare-gates or seqand-gates, respectively. For the spare-gates, we see exactly this number of compositions yielding 9 states and reductions to 5 states. These intermediate models correspond to the composition of the spare-gate with the two successors and the reduced model. For the seqand-gates, we see that we start with 8 steps, but DFTCalc uses twice as many compositions (and reductions). In both cases, only after the composition of all spare/seqand-gates, we compose them via the top-level or-gate. The significant rise in time consumption is thus due to the high overhead of these initial steps.

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert \mathbf{S_n}\rvert_{\text{bs}}}{\lvert \mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
rc-rewr-wsp	14	9	185	4.7		12		32		
rc-rewr-seqand	14	9	185	4.7	1	12	1	52		0.6

Table 6.5.: DFTCalc performance - spare-gates versus seqand-gates.

N-ary gates and time/memory trade-offs First, we notice that the memory consumption of intermediate models is noticeably larger for or-gates with many inputs. If we aim solely for a memory reduction, it thus makes sense to use Rewrite rule 5 in the reverse direction to split large or-gates in small or-gates. In Figure 6.9, we show three different, equivalent fault trees, consisting of eight basic events and one, two or three or-gates, respectively. We assume that all basic events have the same failure rate of 0.1.

# states	# transitions	memory size (KB)
9	16	3.0
5	8	3.0
9	16	3.0
5	8	3.0
9	16	3.0
5	8	3.0
9	16	3.0
5	8	3.0
185	869	4.7
18	76	3.3
141	390	3.8
12	34	3.4

(a) Using spare-gates.

# states	# transitions	memory size (KB)
8	13	3.0
7	11	3.1
8	13	3.0
5	8	3.0
8	13	3.0
7	11	3.1
8	13	3.0
5	8	3.0
8	13	3.0
7	11	3.1
8	13	3.0
5	8	3.0
8	13	3.0
7	11	3.1
8	13	3.0
5	8	3.0
185	869	4.7
18	76	3.3
141	390	3.8
12	34	3.4

(b) Using seqand-gates.

Table 6.6.: The size of the intermediate models.

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert\mathbf{S_n}\rvert_{\text{bs}}}{\lvert\mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
or-8be	9	8	109	4.7		4		17		
variant B	10	8	93	3.8	1.2	4	1	21		0.81
variant C	11	8	67	3.4	1.4	4	1	30		0.56
or-16be	9	16	153	7.9		4		45		
variant B	10	16	113	5	1.6	4	1	36		1.25
variant C	11	15	87	4.3	1.8	4	1	45		1

Table 6.7.: Results of splitting an or-gate with 8/16 BEs as successors in different ways.

We use the single or-gate as baseline and compare the computation of the reliability with mission time 1 by DFTCalc. The results are depicted in the topmost entries in Table 6.7. We again use the standard table with the same entries as before. We observe that the peak model size is indeed considerably (almost a factor two) larger in the baseline. However, the baseline model is synthesised much faster. We can explain this by the reduced number of composition- and minimisation steps executed, while the overall models are in all variants small enough to fit in processor-near cache memory.

We also see this behaviour when replacing all or-gates by and-gates (Table 6.8). We observe that the intermediate models become somewhat larger, which is easily explained by the fact that the failure of a successor is not longer enough to cause a top-level failure. This is also reflected by the fact that the resulting models for and-gates are significantly larger.

The used model is very simple, amongst others because all successors are basic events and the intermediate models keep so small. We therefore replace the basic events by:

- `or-8and2be`: And-gates with two isolated basic events as successors.
- `or-8and4be`: And-gates with four isolated basic events as successors.
- `or-8and2and4be`: And-gates with two isolated and-gates with four isolated basic events as successors.

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert\mathbf{S_n}\rvert_{\text{bs}}}{\lvert\mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
and-8be	9	8	3856	60.7		11		17		
variant B	10	8	362	6.2	10	11	1	21		0.81
variant C	11	8	157	4	15	11	1	30		0.56

Table 6.8.: Results of splitting and and-gate with 8 BEs as successor in different ways

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\text{Mem}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert\mathbf{S_n}\rvert_{\text{bs}}}{\lvert\mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
or-8and2be	17	16	770	9.9		12		68		
variant B	18	16	273	5.5	1.8	12	1	77		0.9
variant C	19	16	213	4.3	2.3	12	1	83		0.8
or-8and4be	34	32	6888	51.1		168		145		
variant B	34	32	2666	23.2	2.2	168	1	148		1
variant C	35	32	3961	28.2	1.8	168	1	155		1
or-8and2and4be	89	64	317344	2364.3		6438		620		
variant B	90	64	219786	1973.8	1.2	6438	1	611		1
variant C	91	64	329346	2402.8	1	6438	1	625		1

Table 6.9.: Results of splitting an or-gate with 8 subtrees in different ways.

The obtained results are displayed in Table 6.9. For smaller subtrees, the results are as before. We notice that especially variant C scales much better when we increase the size of the subtrees. With about eight basic events which have to fail in the subtree, the memory consumption of the flattened or surpasses the variants. Furthermore, for large fault trees, the time spent in the top gate(s) has relatively minor impact, and the speed up drops with the higher memory consumption for constructing the IMC for the or-gate with 8 children. It is important to notice that here, the subtrees are all symmetric, which has a substantial influence on the effect of the intermediate reductions, which increases the performance of DFTCalc on variants B and C.

We conclude that it may be beneficial for memory consumption to avoid gates with a high number of successors — but that this may come at potentially high computation time costs. Among others, it crucially depends on the size of the subtrees. It seems beneficial to always split gates with a very high number of successors. Based on the effects earlier, it is important to take the structure of the successors into account when applying the split — although we did not discuss exactly how. Furthermore, we did not discuss the effect of different subtrees, nor the effect of elements shared among different successors. Quick tests indicate that these all influence the choices of the underlying heuristic.

DFTCalc benchmark performance As it is hard to find general advise for good rewriting measures that improve the performance of DFTCalc, we take a closer look at the obtained baseline results for different benchmark instances from the generated sets. We recall that tables with the full results are given in Appendix A on page 189. Here, we discuss the observations for two benchmark sets. For the other benchmark sets, results are either as expected and/or in line with the here presented effects.

HECS.

We consider eight different families (three binary variables, all combinations) and show the performance for each number of computers in the system. The first variable is whether the overall system fails only if all computers fail (and) or already with the first failure (or). This reflects whether the top-level element is an AND or an OR. For just one computer system, these variables coincide.

The second variable reflects whether we consider the single-memory interface whose failure causes a failure of all memory elements (1), or if we consider the full memory interface specification with two interfaces (2). The third variable reflects whether we use the shared power (up) or not use it (np).

We regard the running time as depicted in Figure 6.10a on page 173. The number of elements and basic events in the DFT are a constant factor of the number of systems, yet the runtime in each subfamily grows exponentially. Whether the top-level is an AND or an OR does not influence the model construction time. Whereas the effect for smaller models is neglectable, for larger models we see a significant influence of the shared power source, which prevents early pruning of the intermediate models. The complex memory interface has a large impact on the construction time, more than what is directly justified by the number of elements.

We also regard the final model size, depicted in Figure 6.10b. We notice that the final model sizes of the or-variants is mostly larger than the and-variant, which does not agree with theoretical bounds. The effect can however be explained by what seems a more radical reduction of the state space in the and-case. We furthermore notice the higher numbers of the and/2/np combination and the remarkable figures for the single system case.

The intermediate model size (Figure 6.10c) roughly correspond to the final model sizes, being an order of magnitude larger. These models are constructed near the end of the composition, when the subtrees are handled. For small systems, the overhead is significantly larger.

We do not depict the effect of different voting gates as top-level elements. From the data in Appendix A on page 189 we deduce that the larger (theoretical) model size of higher thresholds, combined with a less aggressive reduction yields slightly longer running times and model sizes.

RC.

For the RC benchmark set, we start with the simple controller and scale the number of sensors and barriers. We depict the DFTCalc performance in three groups of three graphs. Each group depicts (from left to right) the run time, the maximal intermediate model size and the final model size.

In the first group, with the performance depicted in Figure 6.11 on page 174, we took as many sensors as barriers ($n = 1$ to $n = 5$). The number of elements and basic events is now roughly a constant factor of the number of sensors and barriers. The running time roughly scales with the number of elements (to be precise, doubling the number of elements causes tripling the running time). The intermediate model size starts growing exponentially, but soon the heuristic takes an order which yields a tremendously smaller memory footprint. The final model size grows roughly quadratic, which corresponds to the growth of the theoretical bounds.

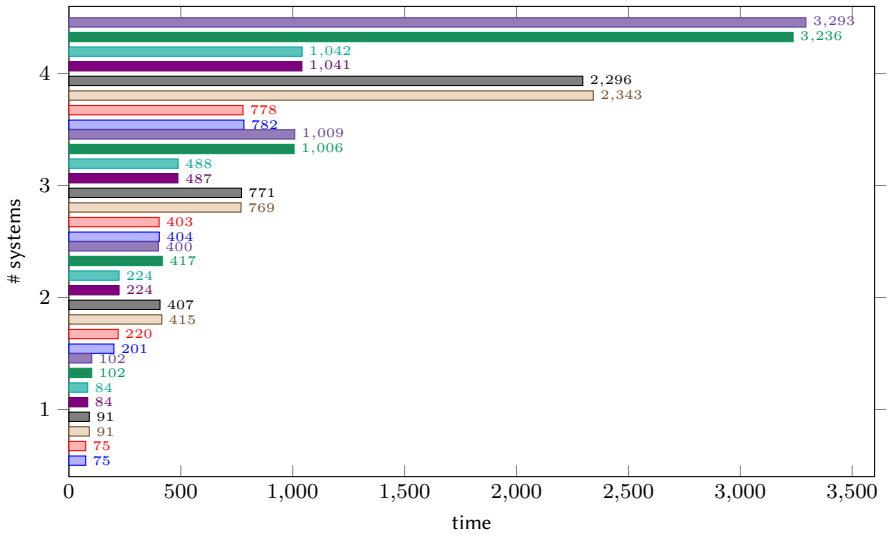
In Figure 6.12 on page 174, we examine the effect of a higher number of sensors (red, squares) versus a higher number of barriers (blue, circles) by using only one barrier or sensor, respectively. While the sensors add more elements and basic events to the DFT, the more complex barrier subtrees yield a higher running time. The higher number of basic events seems to cause a slightly larger final model size (although aggressive reductions should yield equivalent model sizes.) The effect is more dramatic for the intermediate sizes. As for the combined scaling, the intermediate sizes drop after 4 sensors or barriers, respectively, although the top-level or-gate has more children with three barriers and sensors than with one sensor (barrier) and three barriers (sensors).

In Figure 6.13 on page 174, we examine what happens if we replace the controller by a HECS DFT. We consider again an identical number of sensors and barriers. The running time rises slightly slower, which is not surprising as the effect of the sensors and barriers is initially quite small. Interestingly, we see that also the final model size drops sharply, before rising again. The intermediate model size has its peak earlier than with the simple controller. For larger models, we then have a largest intermediate model which corresponds to the original final model.

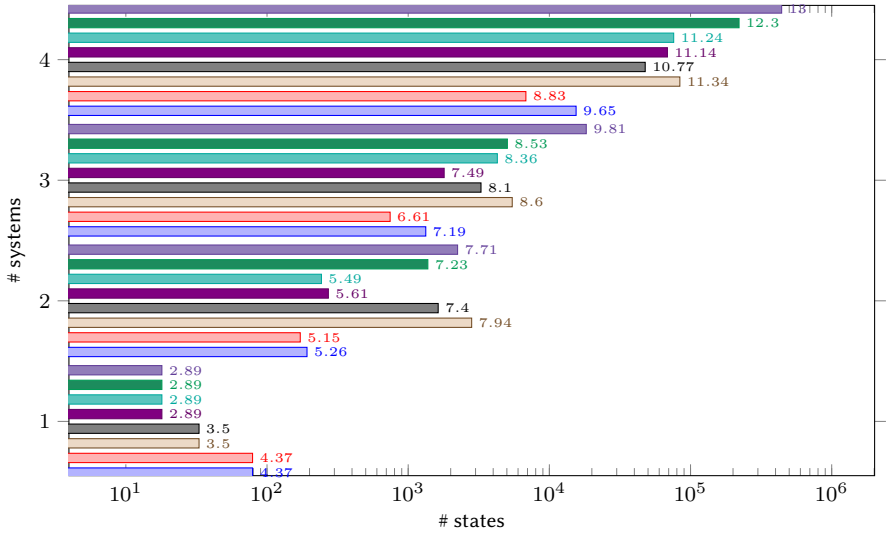
6.3.3. The effect of rewriting

We start this by giving an overall impression, after which we continue with more details.

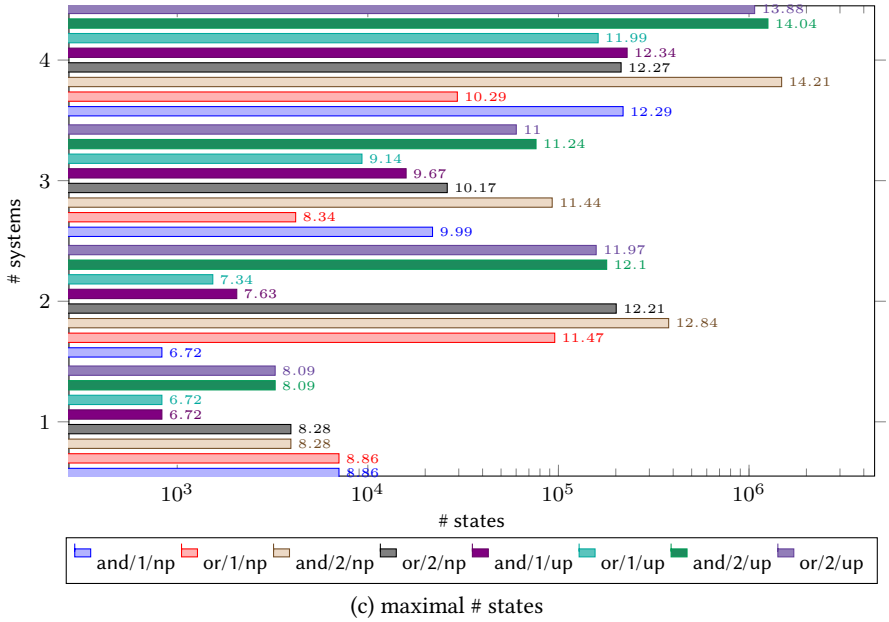
Summary We present a graphical overview of the results in Figure 6.14 on page 175. In Figure 6.14d on page 175 we list the seven benchmark sets, each in a row. Between parentheses, the size of the instances we used is given. The second and third columns give the number of computed instances within time and memory limits, for the baseline and the rewritten version, respectively. Columns four and six give the total time required to solve the baseline and the rewritten instances, respectively. Column five gives the time required to compute the results for the instances that



(a) running time in seconds.



(b) final # states.



(c) maximal # states

Figure 6.10.: DFTCalc performance on baseline HECS.

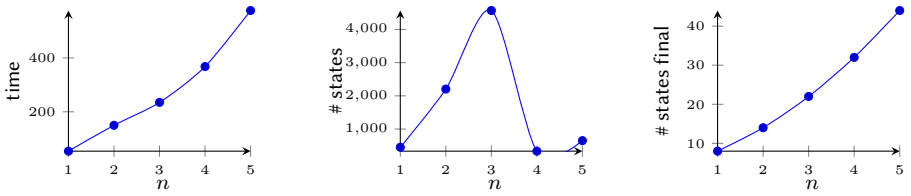


Figure 6.11.: DFTCalc performance on baseline RC/sc.

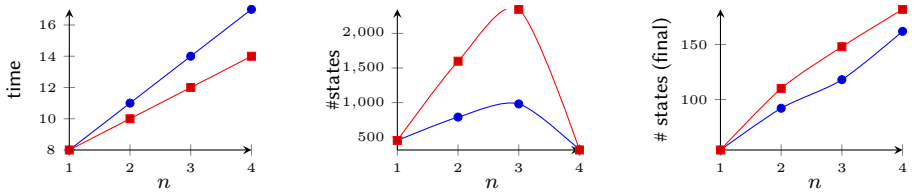


Figure 6.12.: DFTCalc performance on baseline RC/sc - effect of sensors versus barriers.

could also be solved on the baseline input. The last column (seven) gives the ratio of the number of elements in the original and the rewritten instances. We see that out of 183 instances we used for benchmarking, 174 could be solved after rewriting. Without rewriting, only 125 could be solved. While we can eliminate about 30 % of the elements with the rewrite procedure, the average speed-up for the instances which could already be solved is a factor 3. In particular, using rewriting, we can solve the 49 additional (hard) benchmarks while using less time than the original baseline. We depict the single instances from the scalable benchmark sets in three scatter plots.

In Figure 6.14a on page 175, the run time of DFTCalc is plotted. A dot in the scatter plot corresponds to an instance of either the HECS, RC, MCS or SF benchmark set. The value along the x-axis gives the run time of DFTCalc without rewriting. Along the y-axis, the run time after rewriting is given. For both axes, we use a log-scale. At the upper and right end of the axes, entries corresponding to a timeout (2 hours) or memory-out (8 GB) are given. Any dot above the diagonal thus corresponds to an improved run time. A point above the dashed line corresponds to speed-up of more than a factor 10. We see that for these benchmarks, we almost always get speed-ups. Overall, these speed-ups become bigger on larger instances.

In Figure 6.14b, the memory footprint of the intermediate models is plotted. We use again a log-log scale. Here, any dot above the diagonal corresponds to an instance where the memory consumption is reduced. Above the dashed line, the memory consumption is reduced by more than a factor 10. Some small RC instances as well as various SF instances yield a higher memory consumption, however, for the majority of instances, the memory consumption is reduced, even up to a factor 1000. The memory consumption is one of the major bottlenecks. Therefore, a reduction here has a lot of impact.

In Figure 6.14c, the final model size of the instances is plotted (on a log-log scale). Any dot above the diagonal corresponds to an instance where rewriting yields a reduced final model size. Any dot above the dashed line corresponds to a reduction of more than a factor 10. While for most instances, the final model size is not much reduced by rewriting, the final model size is drastically reduced for most instances of HECS and some of SF. This smaller final model size causes speed-ups in the computation of quantitative measures, which becomes important if more and more complex quantitative analyses are requested.

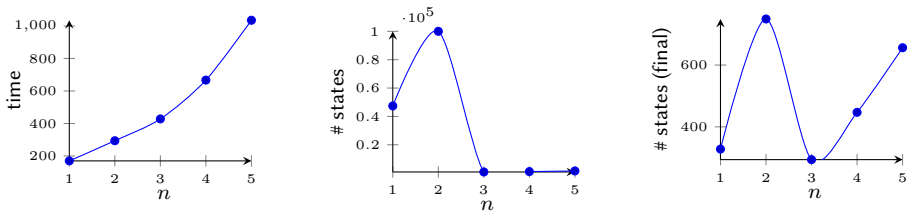
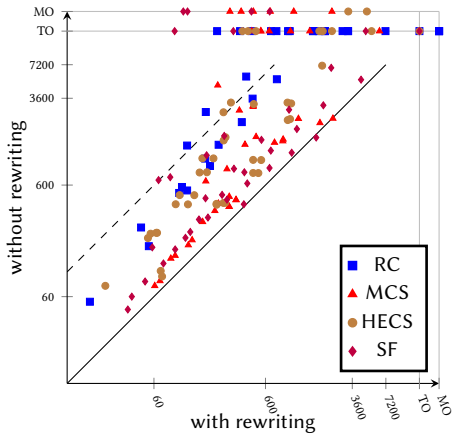
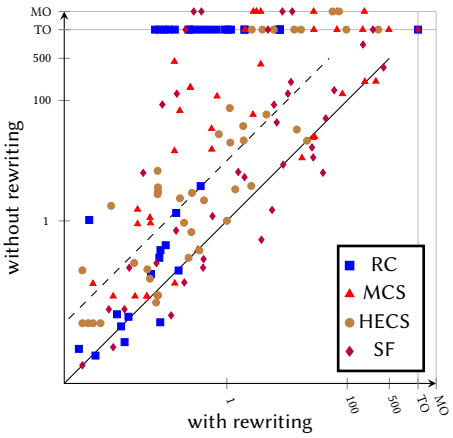


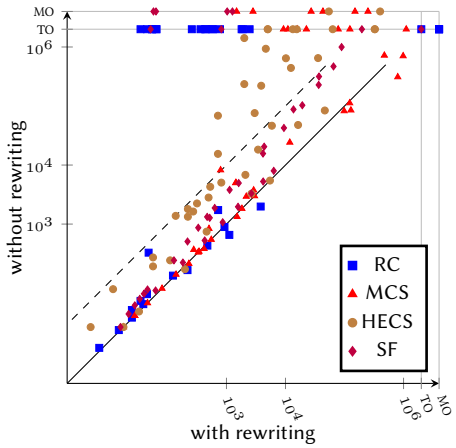
Figure 6.13.: DFTCalc performance on baseline RC/hc.



(a) Run time



(b) Memory footprint



(c) Final memory size

	solved		Σ time (h)			red.
	bs	rw	bs	rw$^{(1)}$	rw$^{(2)}$	$\frac{\lvert V\rvert_{\text{rw}}}{\lvert V\rvert_{\text{bs}}}$
HECS(44)	34	43	11.8	3.3	9.1	1.4
MCS(44)	30	43	9.3	3.7	8.2	1.1
RC(36)	15	31	7.3	5.1	9.3	2.1
SF(39)	31	38	10.1	5.3	7.1	1.5
MOV(8)	3	7	2.3	0.6	0.7	3.4
HCAS(8)	8	8	0.4	0.3	0.3	1.2
SAP(4)	4	4	0.1	0.1	0.1	1.7
total(183)	125	174	41.3	18.4	34.8	1.6

(1) time on instances solved by all.
(2) time on all instances solved.

(d) summary.

Figure 6.14.: Overview of the experimental results on four different benchmark sets.

HECS We show the scaling behaviour of HECS for a growing number of systems and for different configurations in Figures 6.15 to 6.18 on page 176 and on page 177. For each configuration, we give a series of three plots. All x-axes are labelled with the scaled variable. The first plot presents the run time (on the y-axis). The second plot gives the memory footprint in KB of the largest intermediate model along the y-axis. The last plot gives the number of states in the final model. In each plot, 4 data sets are plotted. Blue circles and a dashed line correspond to the baseline of the standard variant, purple rectangles an a dashed line correspond to the baseline of an alternative variant. The rewritten instances are displayed by a brown circle and a solid line for the standard variant and a red diamond with a solid line for the alternative variant.

Remark 39. The following plots are all much alike. The discussion given above holds analogously for all further plots constructed.

In Figure 6.15, we consider a configuration where one computer system suffices to keep the the overall system operational. Each computer system has a single memory unit interface. The standard variant does not contain a failable power-unit while the alternative does. We see that the influence of the power-unit is marginal. For the rewritten instances, the extra power unit actually improves performance as the heuristic performs better with the power unit.

In Figure 6.16, we change the configuration by adding the second memory unit interface. This extra memory interface improves performance for both the base and the rewritten instances. The model size of the rewritten version scales well, which yields a good improvement in terms of computation size and intermediate model size.

In Figure 6.17 and Figure 6.18 on page 177, the configurations above are changed by letting the system fail as soon as the first subsystem fails. The heuristic for one memory interface in the standard variant performs worse on the rewritten version, which is likely due to the large number

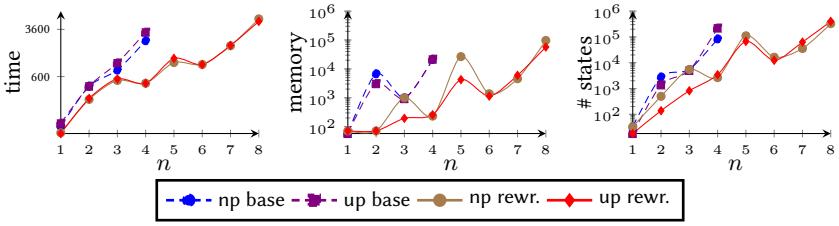


Figure 6.15.: Effect of rewriting on HECS (top $\in$ AND, 1 MUI, n = # systems, np/up = no/use power).

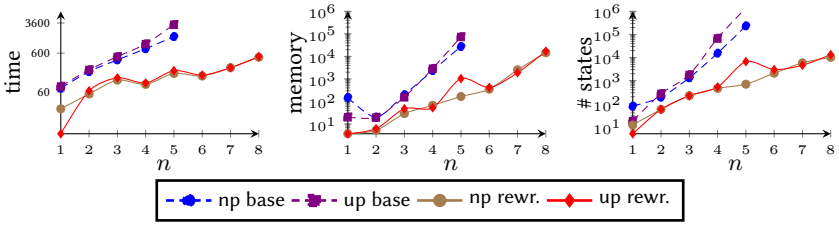


Figure 6.16.: Effect of rewriting on HECS (top $\in$ AND, 2 MUIs, n = # systems, np/up = no/use power).

of successors of the top-level element. On the other instances, rewriting works out good, especially with two memory interfaces.

MCS In Figure 6.19 on page 177, we display the performance for MCS for a growing number of computing modules in a system. We depict two variants, one with the single power supply, one with double power. First of all, the intermediate model size grows exponential, as expected. The double power variant has a — constant — larger state space (theoretically twice as big). Rewriting does not affect this, it is not able to remove many states. The drop at $n = 8$ remains unexplained. While the exponential growth of the run time is initially governed by the same rate, we see that for each variant, at some point the run time rises sharply. For the double power, this point is reached very early. Here, the internal heuristic copes with large intermediate models early on. For slightly larger systems, also the single-power variant runs into time outs, which is mainly due to the massive memory consumption. Rewriting reduces the memory consumption drastically, and as the heuristics for the rewritten variants of single and double power seem to agree, also the performance is almost equivalent. The oscillating memory footprint is easily explained by the way uneven computing modules share spare components.

RC For RC, we consider two sets of configurations. In Figure 6.20 we have a single sensor unit and a variable number of barriers, while in Figure 6.21 on page 178, there is one barrier and a variable number of sensor units. For both sets, we consider a standard variant with a simple controller and an alternative with the hecs-system as a controller. We see a perfect exponential blow-up in run-time. Before rewriting, increasing the number of sensors has a larger effect than increasing the number of barriers. After rewriting, both variables have approximately the same effect. Intermediate and final model size are chaotic, which indicates that the heuristic could be improved here.

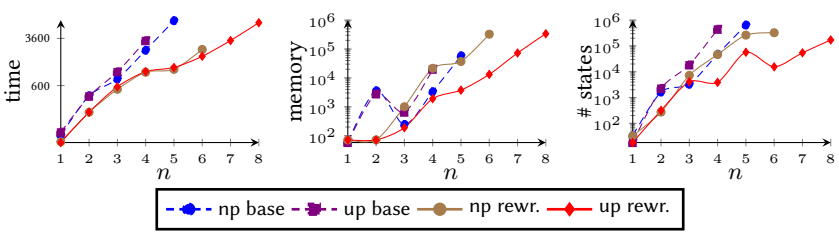


Figure 6.17.: Effect of rewriting on HECS (top $\in$ OR, 1 MUI, n = # systems, np/up = no/use power).

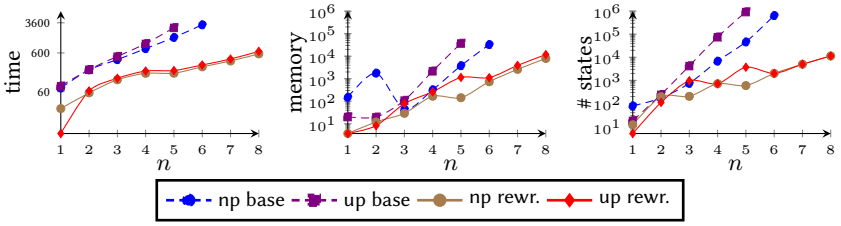


Figure 6.18.: Effect of rewriting on HECS (top $\in$ OR, 2 MUIs, n = # systems, np/up = no/use power).

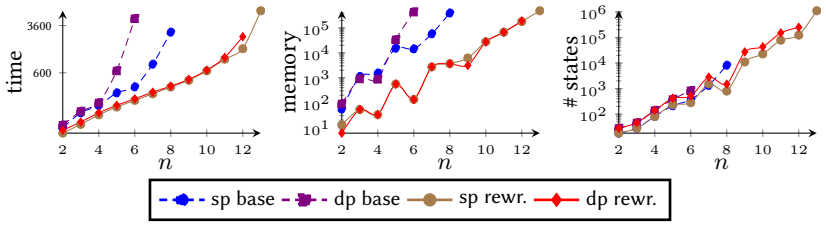


Figure 6.19.: Effect of rewriting on MCS (n = # CMs, sp/dp = single/double power).

SF In Figure 6.22 on page 178, we show the DFTCalc performance for a growing number of filters in the system. Rewriting these instances has only a minor effect. For the memory consumption, the gates with the increased number of elements even cause a penalty. In Figure 6.23, we show the DFTCalc performance for a growing number of sensors in the system. For only one filter, the effect of rewriting is huge, as we reduce n gates to 1 gate with n children, thereby drastically reducing the number of gates that have to be composed. For two filters, this effect is not as large, as the state space explosion kicks in (due to the combination of degraded states of the sensors), and the memory consumption for the overall system has a larger impact on the performance.

SAP For the SAP benchmark, we see that the instances only differ in the which elements are a constant failure and which are fail-safe. For the original input, this yields four times the same performance, as the model construction is identical up to a last small step. With preprocessing, we see major speed-ups as large parts of the DFT can be eliminated by propagating the constant failures.

HCAS The HCAS benchmark instances are too small to really learn lessons from. As the structure is very similar to HECS and RC, we get equivalent results.

Movares The Movares benchmarks are largely static, with often large sets of basic events beneath an or-gate, which can be effectively reduced. In some of the instances the dynamic part is big enough to dominate the memory footprint - on those instances, the effect of rewriting is smaller.

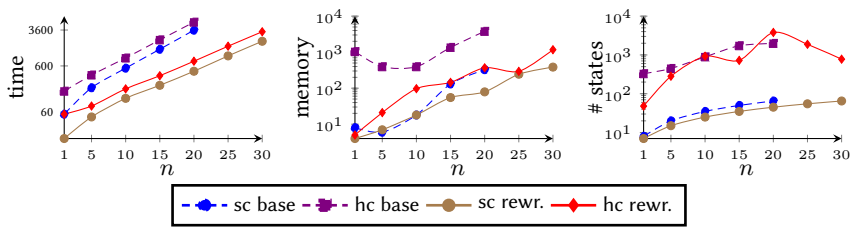


Figure 6.20.: Effect of rewriting on RC with 1 sensor (n = # barriers, sc/hc = single/HECS contr.).

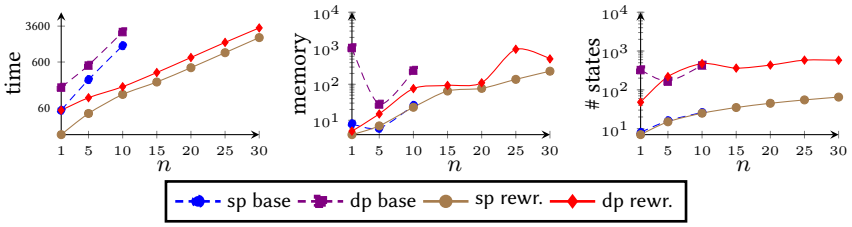


Figure 6.21.: Effect of rewriting on RC with 1 barrier (n = # sensors, sc/hc = single/HECS contr.).

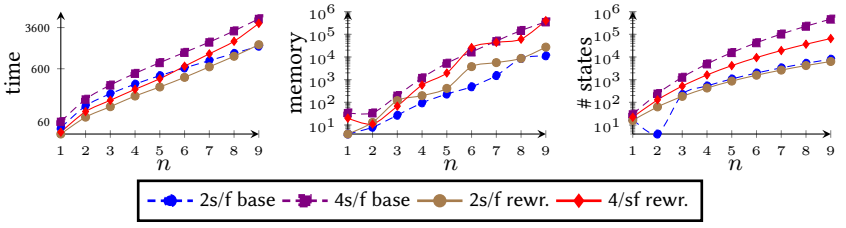


Figure 6.22.: Effect of rewriting on SF (n = # filters s/f = sensors per filter).

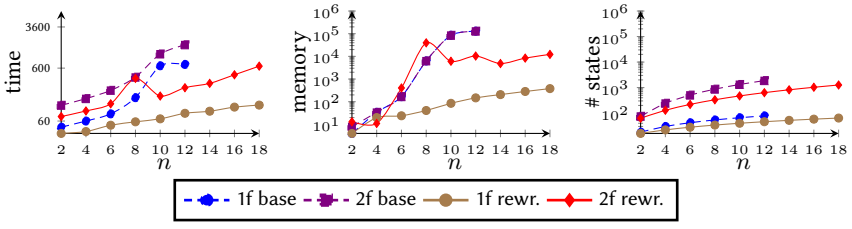


Figure 6.23.: Effect of rewriting on SF (n = # sensors/filter, f = filters).

7. Conclusion

7.1. Summary

In the thesis, we presented a framework to simplify DFTs, driven by the idea that simpler DFTs can be analysed faster. We transform a DFT into another "simplified" DFT by using graph rewriting. As the analysis of DFTs suffers from the omni-present state-space explosion problem and the time to analyse grows exponential with the size (or complexity) of the DFT, a marginally "simpler" (smaller) DFT may be analysed orders of magnitude faster. In order to benefit from this simplification, analysis of a "simplified" DFT has to yield the same results as analysis of the original DFT. To this end, a notion of equivalence and a way of proving two DFTs equivalent was required. In fact, we searched for simple characterisations of the behaviour of a DFT to cater for easy-to-understand proofs.

This quest led us to several intricacies of the existing DFT semantics, presented in Chapter 3 on page 27. With these intricacies in mind, a new DFT semantics was defined along with several characterisations, presented in Chapter 4 on page 71. The rewrite framework as presented in Chapter 5 on page 113 then reduced DFT rewriting to simple graph rewriting. The full complexity of the proof obligations is hidden in the framework, many rewrite rules can be proven by merely showing that they fulfil intuitive criteria. We found that with the developed framework, many case studies can be reduced and that this has a major influence on the analysis time, as discussed in Chapter 6 on page 161.

In conclusion, while the many semantic intricacies make correctness proofs a tedious task, the outcome — major speedups and significantly larger DFTs within time and memory limits can be analysed after graph rewriting — supports our initial hypothesis.

Contribution We list the main contributions of the thesis in order of appearance:

- We give a comprehensive overview on Dynamic Fault Trees, with a focus on their semantics. Based on the discussion of several intricacies, we characterise - to the best of our knowledge - all DFT semantics from the literature.
- We present an extensive literature review regarding existing case studies. We provide a benchmark suite based on seven different existing case studies. Four families are generated by a script and allow for arbitrary scaling the benchmarks along several dimensions.
- We introduce a new semantics defined on a non-restrictive syntax for DFTs and include several characterisations thereof, thereby showing that it captures the intuition of the DFT semantics. We show that the semantics are suitable to define advanced concepts such as partial order reduction.
- We present a framework for rewriting DFTs - based on the reduction of DFTs to ordinary graphs. We show that, due to the context sensitivity, rewriting DFTs is non-trivial. We present theorems which uncover local criteria on rewrite rules which suffice to show that the application of such a rule preserves the measures-of-interest. With these theorems, correctness of rewrite rules is easily deduced. We illustrate this on a total of 28 rewrite rules.
- We implemented prototypical tool-support build around Groove yielding a fully automatised tool-chain for rewriting DFTs.
- We show the practical relevance of the rewriting procedure on over 200 benchmark instances by comparing the analysis in DFTCalc of the original and the corresponding simplified DFT. In almost all cases, rewriting yields a speed-up, on several cases this speed-up is more than a factor 10. The memory reduction is drastically reduced for most benchmark instances - up to three orders of magnitude. The state space of the underlying Markov chain is reduced in

most cases. Based on the scalable benchmarks, we show that rewriting has a super-linear effect on the performance of DFTCalc. The detailed results from the experiments helped the developers of DFTCalc to drastically improve its performance.

7.2. Discussion and Future Work

We see future work in three more-or-less orthogonal directions, namely *DFT extensions*, *state-space generation*, and *proof and rewrite automatisation*. Besides these DFT-related challenges, we are interested to widen the approach also to attack trees [MO06] and other (graphical) formalisms.

DFT extensions DFT extension entails extension of the supported gate types as discussed in Section 4.5 on page 110. It also entails more liberal forms of sharing (items can be claimed by up to n spare gates) and dedicated gates for activation throughput (as discussed in Section 3.3.4 on page 40). Although such extensions make the semantics more complex, they reduce the number of "hacks" that have to be applied in order to faithfully model specific behaviour. We conjecture that this mostly requires work regarding the rewrite framework and correctness thereof - the impact on the semantics itself should be minor.

In another direction, it may be worthwhile to replace some existing gates by some other automaton and consider the Cartesian product of a (static) fault tree and such an automaton (which may encode activation and/or sequence enforcing), to prevent the considerable amount of hacks already present in DFTs.

Furthermore, it would be interesting to add repairs and non-coherent gates to the semantics. This however, requires thoroughly rethinking some of the assumptions we made in this thesis - many proofs rely on the monotonicity of the various mechanisms (failure, claiming, etc.). One approach would be extending the event traces to include failure and repair events.

Another direction is the consider more general properties and whether they are conserved by the rewriting. One of the eminent reasons to use Bayesian networks as underlying model seems the native support of those networks to a number of interesting properties. We believe that many interesting properties (e.g., probability that the system fails without occurrence of a specific basic event, mean time between the first component failure and a system failure) can be expressed elegantly using the underlying Markov automaton.

Using Markov chains furthermore calls for embracing recent results where failure rates are represented by intervals or symbolically. This seems natural in the context of DFTs, where failure rates are inherently inexact and can, during design time, be varied (reflecting the reliability levels of available components), and would ultimately allow us to synthesise cost-minimal combinations of components which meet the required system reliability levels. Construction and analysis of these models is more expansive, which makes effective preprocessing even more valuable than in the current setting.

State-space generation Future work in state space generation is largely motivated by the observations in Chapter 6 on page 161. While the IOIMC semantics scales well, we see some problems with its implementation DFTCalc. First, some gates have IMC representations which are larger than expected. Second, the creation and composition of the IMCs causes much overhead, which yields a slow state-space generation for small models. We see two major points here. First, state space generation in DFTCalc can be improved much based on the characterisations given in this thesis, such as failure paths and partial order reduction. Second, the denotational semantics in Chapter 4 on page 71 can be used to generate the underlying Markov automaton directly. With the help of the partial order reduction (which can be extended to other forms of symmetry) and some of the available characterisations, the underlying MA can be kept smaller than the IOIMC counterpart. Another benefit of this is that such a tool would yield a Markov automaton which reflects the — in our opinion — easier to understand semantics. We do not think that there is a silver bullet here - therefore, a hybrid approach in which IMCs for subDFTs are generated with the denotational semantics and then combined by the IOIMC semantics may be worth the effort. Furthermore, the state space generation would benefit from tailored heuristics in the rewriting process to fully unfold the potential simplification, especially if combined with some dedicated gates (like

the sequantial-and). Moreover, using GSPNs with their native support for true concurrency as the underlying stochastic model seems a promising alternative to the usage of Markov chains.

Automatised proofs and rewrite rules While many rewrite rules are easy and straightforward to prove, defining the context restrictions to be as permissive as possible is a tedious and error-prone task. The help of (semi)-automated theorem provers could considerably improve the theory. Likewise, the translation of the rules to Groove is tedious and error-prone, and a tool to automatically translate the DFT rules into a Groove grammar would be of great value.

Bibliography

[IEC60050] "*Fault tree analysis (FTA)*". Norm IEC 60050:2006. 2007 (cited on pages 37, 48).

[ABBG+13] F. Arnold, A. Belinfante, F. van der Berg, D. Guck, and M. Stoelinga. "*DFTCalc: A Tool for Efficient Fault Tree Analysis*". *Proc. of SAFECOMP*. Volume 8153. LNCS. Springer, 2013, pages 293–301 (cited on pages 3, 57, 69).

[AD00] R. B. Ash and C. Doléans-Dade. "*Probability and Measure Theory*". Academic Press, 2000 (cited on pages 7, 9, 12).

[AD94] R. Alur and D. L. Dill. "*A theory of timed automata*". *Theoretical Computer Science* 126.2 (1994), pages 183–235 (cited on page 2).

[AEHH+99] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. "*Graph transformation for specification and programming*". *Science of Computer Programming* 34.1 (1999), pages 1–54 (cited on page 3).

[Apo67] Apollo 204 Review Board. "*Apollo 204 Review Board: Final Report*". 1967 (cited on page 1).

[Bal07] G. Balbo. "*Introduction to Generalized Stochastic Petri Nets*". *Formal Methods for Performance Evaluation*. Volume 4486. LNCS. Springer, 2007, pages 83–131 (cited on page 69).

[BB96] N. Balakrishnan and A. P. Basu. "*Exponential Distribution: Theory, Methods and Applications*". CRC Press, 1996 (cited on pages 8, 9).

[BC04] A. Bobbio and D. Codetta-Raiteri. "*Parametric fault trees with dynamic gates and repair boxes*". *Proc. of RAMS*. 2004, pages 459–465 (cited on pages 4, 68).

[BCCK+10] M. Bozzano, R. Cavada, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and X. Olive. "*Formal verification and validation of AADL models*". *Proc. of ERTS*. 2010 (cited on pages 2, 60).

[BCKN+09] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, and M. Roveri. "*The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems*". *Proc. of SAFECOMP*. Volume 5775. LNCS. Springer, 2009, pages 173–186 (cited on page 60).

[BCS07a] H. Boudali, P. Crouzen, and M. Stoelinga. "*A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains*". *Proc. of ATVA*. Volume 4762. LNCS. Springer, 2007, pages 441–456 (cited on pages 58, 60, 69).

[BCS07b] H. Boudali, P. Crouzen, and M. Stoelinga. "*CORAL - a tool for compositional reliability and availability analysis*". *Proc. of ARTIST*. 2007 (cited on page 69).

[BCS07c] H. Boudali, P. Crouzen, and M. Stoelinga. "*Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains*". *Proc. of DSN*. IEEE Computer Society, 2007, pages 708–717 (cited on pages 4, 40, 69, 113).

[BCS10] H. Boudali, P. Crouzen, and M. Stoelinga. "*A Rigorous, Compositional, and Extensible Framework for Dynamic Fault Tree Analysis*". *IEEE Transactions on Dependable Secure Computing* 7.2 (2010), pages 128–143 (cited on pages 36, 50, 69).

[BD05a] H. Boudali and J. B. Dugan. "*A new Bayesian network approach to solve dynamic fault trees*". *Proc. of RAMS*. 2005, pages 451–456 (cited on pages 4, 60).

[BD05b] H. Boudali and J. B. Dugan. "*A discrete-time Bayesian network reliability modeling and analysis framework*". *Reliability Engineering & System Safety* 87 (2005), pages 337–349 (cited on pages 4, 56, 58, 67, 68).

[BD06] H. Boudali and J. B. Dugan. "*A continuous-time Bayesian network reliability modeling and analysis framework*". *IEEE Transactions on Reliability* 55.1 (2006), pages 86–97 (cited on pages 4, 67, 68).

[BFGP03] A. Bobbio, G. Franceschinis, R. Gaeta, and L. Portinale. "*Parametric fault tree for the dependability analysis of redundant systems and its high-level Petri net semantics*". *IEEE Transactions on Software Engineering* 29.3 (2003), pages 270–287 (cited on pages 33, 68).

[BHHK03] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. "*Model-checking algorithms for continuous-time Markov chains*". *IEEE Transactions on Software Engineering* 29.6 (2003), pages 524–541 (cited on page 13).

[BK08] C. Baier and J.-P. Katoen. "*Principles of Model Checking*". The MIT Press, 2008 (cited on pages 7, 9, 13, 14, 77, 102, 103, 108).

[BKKM14] C. Baier, J. Klein, S. Klüppelholz, and S. Märcker. "*Computing Conditional Probabilities in Markovian Models Efficiently*". *Proc. of TACAS*. Volume 8413. LNCS. Springer, 2014, pages 515–530 (cited on page 15).

[BNS09] H. Boudali, A. Nijmeijer, and M. Stoelinga. "*DFTSim: A Simulation Tool for Extended Dynamic Fault Trees*". *Proc. of ANSS*. Society for Modeling and Simulation International, 2009, page 31 (cited on page 67).

[CA80] T. Chu and G. Apostolakis. "*Methods for Probabilistic Analysis of Noncoherent Fault Trees*". *IEEE Transactions on Reliability* R-29.5 (1980), pages 354–360 (cited on page 33).

[CCDM+11a] F. Chiacchio, L. Compagno, D. D'Urso, G. Manno, and N. Trapani. "*An open-source application to model and solve dynamic fault tree of real industrial systems*". *Proc. of SKIMA*. IEEE India, 2011, pages 1–8 (cited on page 67).

[CCDM+11b] F. Chiacchio, L. Compagno, D. D'Urso, G. Manno, and N. Trapani. "*Dynamic fault trees resolution: A conscious trade-off between analytical and simulative approaches*". *Reliability Engineering & System Safety* 96.11 (2011), pages 1515–1526 (cited on page 62).

[CCR08] S. Contini, G. Cojazzi, and G. Renda. "*On the use of non-coherent fault trees in safety and security studies*". *Proc. of ESREL*. Volume 93. 12. Elsevier, 2008, pages 1886–1895 (cited on page 33).

[CDFH93] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. "*Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications*". *Transactions on Computing* 42.11 (1993), pages 1343–1360 (cited on page 68).

[CE82] E. M. Clarke and E. A. Emerson. "*Design and synthesis of synchronization skeletons using branching time temporal logic*". *Logics of Programs*. Volume 131. LNCS. Springer, 1982, pages 52–71 (cited on page 2).

[CH05] L. Cloth and B. R. Haverkort. "*Model checking for survivability!*" *Proc. of QEST*. 2005, pages 145–154 (cited on page 55).

[CHM07] S. Chen, T. Ho, and B. Mao. "*Reliability evaluations of railway power supplies by fault-tree analysis*". *Electric Power Applications, IET* 1.2 (2007), pages 161–172 (cited on page 1).

[Cod05] D. Codetta-Raiteri. "*The Conversion of Dynamic Fault Trees to Stochastic Petri Nets, as a case of Graph Transformation*". *Proc. of PNGT*. Volume 127. 2. 2005, pages 45–60 (cited on pages 4, 57, 69).

[Col03] Columbia Accident Investigation Board. "*Columbia Accident Investigation Board: Report Volume I*". 2003 (cited on page 1).

[Cro06] P. Crouzen. "*Compositional Analysis of Dynamic Fault Trees using IOIMCs*". Master's thesis. University of Twente, 2006 (cited on page 57).

[CSD00] D. Coppit, K. J. Sullivan, and J. B. Dugan. "*Formal semantics of models for computational engineering: a case study on Dynamic Fault Trees*". *Proc. of ISSRE*. IEEE Computer Society, 2000, pages 270–282 (cited on pages 4, 40, 44, 67, 71).

[CY95] C. Courcoubetis and M. Yannakakis. "*The Complexity of Probabilistic Verification*". *Journal of the ACM* 42.4 (1995), pages 857–907 (cited on page 2).

[DBB90] J. B. Dugan, S. J. Bavuso, and M. Boyd. "*Fault trees and sequence dependencies*". *Proc. of RAMS*. 1990, pages 286–293 (cited on page 34).

[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 (cited on pages 56, 58, 59).

[DKH97] F. Drewes, H.-J. Kreowski, and A. Habel. "*Handbook of Graph Grammars and Computing by Graph Transformation*". World Scientific Publishing, 1997. Chapter Hyperedge Replacement Graph Grammars, pages 95–162 (cited on page 17).

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. "*Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*". Springer, 2006 (cited on page 17).

[EFT96] H. Ebbinghaus, J. Flum, and W. Thomas. "*Mathematical Logic*". Undergraduate Texts in Mathematics. Springer, 1996 (cited on page 7).

[Ehr79] H. Ehrig. "*Introduction to the algebraic theory of graph grammars (a survey)*". *Graph-Grammars and Their Application to Computer Science and Biology*. Volume 73. LNCS. Springer, 1979, pages 1–69 (cited on pages 3, 17–20).

[EHZ10a] C. Eisentraut, H. Hermanns, and L. Zhang. "*On Probabilistic Automata in Continuous Time*". *Proc. of LICS*. IEEE Computer Society, 2010, pages 342–351 (cited on page 9).

[EHZ10b] C. Eisentraut, H. Hermanns, and L. Zhang. "*Concurrency and Composition in a Stochastic World*". *CONCUR 2010 - Concurrency Theory*. Volume 6269. LNCS. Springer, 2010, pages 21–39 (cited on page 9).

[Eri99] C. A. Ericson II. "*Fault Tree Analysis - A History*". *Proc. of ISSC*. 1999 (cited on pages 1, 28).

[EWG12] E. Edifor, M. Walker, and N. Gordon. "*Quantification of Priority-OR Gates in Temporal Fault Trees*". *Proc. of SAFECOMP*. Volume 7612. LNCS. Springer, 2012, pages 99–110 (cited on pages 35, 64).

[FGH06] P. H. Feiler, D. P. Gluch, and J. J. Hudak. "*The Architecture Analysis & Design Language (AADL): An Introduction*". Technical report CMU/SEI-2006-TN-011. Software Engineering Institute, Carnegie Mellon University, 2006 (cited on page 60).

[GdRZ+12] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. "*Modelling and analysis using GROOVE*". *Int'l Journal on Software Tools for Technology Transfer* 14 (2012), pages 15–40 (cited on pages 3, 19).

[Gha98] Z. Ghahramani. "*Learning dynamic Bayesian networks*". *Adaptive Processing of Sequences and Data Structures*. Springer, 1998, pages 168–197 (cited on page 67).

[GHHK+13] D. Guck, H. Hatefi, H. Hermanns, J.-P. Katoen, and M. Timmer. "*Modelling, Reduction and Analysis of Markov Automata*". *Proc. of QEST*. Volume 8054. LNCS. Springer, 2013, pages 55–71 (cited on pages 11, 14, 167).

[GKSL+14] D. Guck, J.-P. Katoen, M. Stoelinga, T. Luiten, and J. Romijn. "*Smart railroad maintenance engineering with stochastic model checking*". *Proc. of RAILWAYS*. Volume 104. Civil-Comp Proceedings. Civil-Comp Press, 2014, page 299 (cited on pages 1, 56, 57, 65, 165, 166).

[GMKA14] A. N. Gharahasanlou, A. Mokhtarei, A. Khodayarei, and M. Ataei. "*Fault tree analysis of failure cause of crushing plant and mixing bed hall at Khoy cement factory in Iran*". *Case Studies in Engineering Failure Analysis* 2.1 (2014), pages 33–38 (cited on page 1).

[Goo88] G. V. Goodman. "*An assessment of coal mine escapeway reliability using fault tree analysis*". *Mining Science and Technology* 7.2 (1988), pages 205–215 (cited on page 1).

[Guc12] D. Guck. "*Quantitative analysis of Markov automata*". Master's thesis. RTWH Aachen University, 2012 (cited on pages 11, 15).

[Hal60] P. Halmos. "*Naive Set Theory*". Undergraduate Texts in Mathematics. Springer, 1960 (cited on page 7).

[Hav14] B. R. Haverkort. "*Model Checking for Survivability Evaluation Critical Infrastructures*". Presentation slides. 2014 (cited on page 55).

[Her02] H. Hermanns. "*Interactive Markov Chains: And the Quest for Quantified Quality*". *LNCS* (2002) (cited on pages 13, 16).

[HH12] H. Hafeti and H. Hermanns. "*Model checking algorithms for Markov automata*". *Proc. of AVOCS*. Volume 53. Electronic communications of the EASST. 2012 (cited on page 14).

[HK10] H. Hermanns and J.-P. Katoen. "*The how and why of interactive Markov chains*". *Formal Methods for Components and Objects*. Volume 6286. LNCS. Springer, 2010, pages 311–337 (cited on page 16).

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. "*Introduction to Automata Theory, Languages, and Computation (3rd Edition)*". Addison-Wesley, 2006 (cited on page 7).

[HSSW06] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter. "*GXL: A graph-based standard exchange format for reengineering*". *Science of Computer Programming* 60.2 (2006), pages 149–170 (cited on page 164).

[IEC60050-191] "*International Electrotechnical Vocabulary. Chapter 191: Dependability and quality of service*". Norm IEC 60050-191. 1990 (cited on pages 1, 30).

[ISO 24765] "*Systems and software engineering - Vocabulary*". Norm ISO/IEC/IEEE 24765. 2010 (cited on pages 27, 29).

[JBWD+14] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Hegedüs, M. Herrmannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. M. Rose, S. Wätzoldt, and S. Mazanek. *Science of Computer Programming* 85 (2014), pages 41–99 (cited on page 16).

[KZHH+11] J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. "*The ins and outs of the probabilistic model checker {MRMC}*". *Performance Evaluation* 68.2 (2011), pages 90–104 (cited on page 167).

[Lam04] H. Lambert. "*Use of Fault Tree Analysis for Automotive Reliability and Safety Analysis*". Technical report. 2004 (cited on page 1).

[LXZL+07] D. Liu, W. Xing, C. Zhang, R. Li, and H. Li. "*Cut Sequence Set Generation for Fault Tree Analysis*". *Embedded Software and Systems*. Volume 4523. LNCS. Springer, 2007, pages 592–603 (cited on pages 40, 67).

[MBCD+94] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. "*Modelling with Generalized Stochastic Petri Nets*". 1st. Wiley, 1994 (cited on page 69).

[MCSD99] R. Manian, D. W. Coppit, K. J. Sullivan, and J. B. Dugan. "*Bridging the gap between systems and dynamic fault tree models*". *Proc. of RAMS*. 1999, pages 105–111 (cited on page 67).

[Meu95] J. van der Meulen. "*Definitions for hardware/software reliability engineers*". Simtech B.V., 1995 (cited on page 27).

[MO06] S. Mauw and M. Oostdijk. "*Foundations of Attack Trees*". *Proc. of ICISC*. Volume 3935. LNCS. Springer, 2006, pages 186–198 (cited on page 180).

[MPBC06] S. Montani, L. Portinale, A. Bobbio, and D. Codetta-Raiteri. "*Automatically translating dynamic fault trees into dynamic Bayesian networks by means of a software tool*". *Proc. of ARES*. 2006, pages 804–810 (cited on pages 38, 57, 67, 68).

[MPBC08] S. Montani, L. Portinale, A. Bobbio, and D. Codetta-Raiteri. "*Radyban: A tool for reliability analysis of dynamic fault trees through conversion into dynamic Bayesian networks*". *Reliability Engineering & System Safety* 93.7 (2008), pages 922–932 (cited on pages 38, 67, 68).

[MPBV+06] S. Montani, L. Portinale, A. Bobbio, M. Varesio, and D. Codetta-Raiteri. "*A tool for automatically translating dynamic fault trees into dynamic bayesian networks*". *Proc. of RAMS*. 2006, pages 434–441 (cited on pages 4, 38, 50, 67).

[MRL10] G. Merle, J.-M. Roussel, and J.-J. Lesage. "*Improving the Efficiency of Dynamic Fault Tree Analysis by Considering Gate FDEP as Static*". *Proc. of ESREL*. 2010, pages 845–851 (cited on pages 4, 156).

[MRL14] G. Merle, J.-M. Roussel, and J.-J. Lesage. "*Quantitative Analysis of Dynamic Fault Trees Based on the Structure Function*". *Quality and Reliability Engineering International* 30.1 (2014), pages 143–156 (cited on page 69).

[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 (cited on pages 4, 43, 69).

[MRLV10] G. Merle, J.-M. Roussel, J.-J. Lesage, and N. Vayatis. "*Analytical Calculation of Failure Probabilities in Dynamic Fault Trees including Spare Gates*". *Proc. of ESREL*. 2010, pages 794–801 (cited on page 69).

[MT95] M. Malhotra and K. Trivedi. "*Dependability modeling using Petri-nets*". *IEEE Transactions on Reliability* 44.3 (1995), pages 428–440 (cited on page 57).

[Nat00] National Transportation Safety Board. "*Aircraft Accident Report: In-flight Breakup Over The Atlantic Ocean Trans World Airlines Flight 800*". 2000 (cited on page 1).

[Neu10] M. R. Neuhäußer. "*Model Checking Nondeterministic and Randomly Timed Systems*". PhD thesis. RWTH Aachen University, University of Twente, 2010 (cited on page 13).

[Nor98] J. R. Norris. "*Markov chains*". 2008. Cambridge university press, 1998 (cited on page 13).

[NSK09] M. R. Neuhäußer, M. Stoelinga, and J.-P. Katoen. "*Delayed Nondeterminism in Continuous-Time Markov Decision Processes*". *Proc. of FOSSACS*. Volume 5504. LNCS. Springer, 2009, pages 364–379 (cited on pages 12, 14).

[Pea88] J. Pearl. "*Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*". Morgan Kaufmann Publishers Inc., 1988 (cited on page 67).

[PH08] R. Pulungan and H. Hermanns. "*Effective Minimization of Acyclic Phase-Type Representations*". *Analytical and Stochastic Modeling Techniques and Applications*. Volume 5055. LNCS. Springer, 2008, pages 128–143 (cited on page 33).

[Pie02] B. C. Pierce. "*Types and Programming Languages*". MIT Press, 2002 (cited on page 20).

[Plu02] D. Plump. "*Essentials of Term Graph Rewriting*". *Electronic Notes in Theoretical Computer Science* 51 (2002), pages 277–289 (cited on page 17).

[QS82] J.-P. Queille and J. Sifakis. "*Specification and verification of concurrent systems in CESAR*". *International Symposium on Programming*. Volume 137. LNCS. Springer, 1982, pages 337–351 (cited on page 2).

[RBKS12] A. Rensink, I. Boneva, H. Kastenberg, and T. Staijen. "*User Manual for the GROOVE Tool Set*". Technical report. 2012 (cited on pages 19, 21, 24).

[RH04] M. Rausand and A. Høyland. "*System Reliability Theory: Models, Statistical Methods, and Applications*". Wiley Series in Probability and Statistics - Applied Probability and Statistics Section. Wiley, 2004 (cited on page 1).

[Ros10] S. M. Ross. "*Introduction to Probability Models (Tenth Edition)*". Tenth Edition. Academic Press, 2010 (cited on page 7).

[Roz97] G. Rozenberg, editor. "*Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*". World Scientific Publishing, 1997 (cited on page 16).

[RS14] E. Ruijters and M. Stoelinga. "*Fault Tree Analysis: A survey of the state-of-the-art in modeling, analysis and tools*". Technical Report TR-CTIT-14-14. University of Twente, 2014 (cited on pages 4, 30, 33, 67).

[Saz14] S. Sazonov. "*Property preservation under bisimulations on Markov automata*". Master's thesis. RWTH Aachen University, 2014 (cited on pages 10, 13, 15, 16).

[Sch09] S. Schilling. "*Beitrag zur dynamischen Fehlerbaumanalyse ohne Modulbildung und zustandsbasierte Erweiterungen*". PhD thesis. Universität Wuppertal, 2009 (cited on pages 1, 2, 56, 65, 69).

[SDC99] K. Sullivan, J. B. Dugan, and D. Coppit. "*The Galileo fault tree analysis tool*". *Proc. of FTCS*. 1999, pages 232–235 (cited on page 67).

[Spi92] J. M. Spivey. "*The Z Notation: A Reference Manual*". Prentice Hall International (UK) Ltd., 1992 (cited on page 67).

[Sta03] D. Stamatis. "*Failure Mode and Effect Analysis: FMEA from Theory to Execution*". ASQ Quality Press, 2003 (cited on page 1).

[Sto02] M. Stoelinga. "*Alea jacta est: Verification of Probabilistic, Real-time and Parametric Systems*". PhD thesis. University of Nijmegen, 2002 (cited on page 13).

[TD04] Z. Tang and J. B. Dugan. "*Minimal cut set/sequence generation for dynamic fault trees*". *Proc. of RAMS*. Jan. 2004, pages 207–213 (cited on pages 40, 56, 65, 67).

[The79] The President's Commision on the Accident at Three Mile Island. "*The Accident at Three Mile Island. The Need For Change: The Legacy of TMI*". 1979 (cited on page 1).

[The86] The Presidential Commision on the Space Shuttle Challenger Accident. "*Report of the Presidential Commision on the Space Shuttle Challenger Accident*". 1986 (cited on page 1).

[Tho90] W. Thomas. "*Automata on Infinite Objects*". *Handbook of Theoretical Computer Science (Vol. B)*. MIT Press, 1990, pages 133–191 (cited on page 7).

[Tim13] M. Timmer. "*Efficient Modelling, Generation and Analysis of Markov Automata*". PhD thesis. University of Twente, 2013 (cited on pages 7, 9, 10, 13, 15, 16).

[VCM09] A. Volkanovski, M. Cepin, and B. Mavko. "*Application of the fault tree analysis for assessment of power system reliability*". *Reliability Engineering & System Safety* 94.6 (2009), pages 1116–1127 (cited on page 1).

[VDS99] K. Vemuri, J. B. Dugan, and K. Sullivan. "*Automatic synthesis of fault trees for computer-based systems*". *IEEE Transactions on Reliability* 48.4 (1999), pages 394–402 (cited on pages 57, 58, 60).

[VS02] W. Vesely and M. Stamatelatos. "*Fault Tree Handbook with Aerospace Applications*". Technical report. NASA Headquarters, Washington D.C., USA, 2002 (cited on pages 1, 2, 4, 27, 32, 33, 38, 44, 48, 55, 56, 65, 113, 142).

[Wal09] M. D. Walker. "*Pandora: a logic for the qualitative analysis of temporal fault trees*". PhD thesis. University of Hull, 2009 (cited on pages 4, 69).

[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 page 35).

[WP10] M. Walker and Y. Papadopoulos. "*A Hierarchical Method for the Reduction of Temporal Expressions in Pandora*". *Proc. of DYADEM-FTS*. ACM Press, 2010, pages 7–12 (cited on pages 34, 35, 52, 53, 63, 69).

[XMTY+13] J. Xiang, F. Machida, K. Tadano, K. Yanoo, W. Sun, and Y. Maeno. "*A Static Analysis of Dynamic Fault Trees with Priority-AND Gates*". *Proc. of LADC*. IEEE Computer Society, 2013, pages 58–67 (cited on pages 34, 59).

[YY08] T. Yuge and S. Yanagi. "*Quantitative analysis of a fault tree with priority AND gates*". *Reliability Engineering & System Safety* 93.11 (2008), pages 1577–1583 (cited on pages 34, 56).

[Zam13] E. Zambon. "*Abstract Graph Transformation - Theory and Practice*". PhD thesis. University of Twente, 2013 (cited on pages 16, 19).

[ZN10] L. Zhang and M. R. Neuhüßer. "*Model Checking Interactive Markov Chains*". *Proc. of TACAS*. Volume 6015. LNCS. Springer, 2010, pages 53–68 (cited on page 13).

A. Overview of Results

An asterisk behind the name indicates that the benchmark is included in the scatter plots.

	$\lvert V\rvert$	$\lvert F_{\text{BE}}\rvert$	$\max_i \lvert S_i\rvert$	Mem	$\frac{\text{Mem}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$	$\lvert S_n\rvert$	$\frac{\lvert\mathbf{S_n}\rvert_{\text{bs}}}{\lvert\mathbf{S_n}\rvert_{\text{rw}}}$	t_D	t_G	$\frac{\mathbf{t}_{\text{bs}}}{\mathbf{t}_{\text{rw}}}$
HECS										
h-1-1-1-np*	21	13	7058	151		79		75		
simplified	13	9	123	3	**40.8**	12	**6.6**	22	10	**2.4**
h-1-1-1-up*	24	15	831	20		18		84		
simplified	13	9	123	3	**5.5**	5	**3.6**	5	10	**5.4**
h-1-1-2-np*	25	14	3948	58		33		91		
simplified	23	13	5477	71	**0.8**	33	**1.0**	71	9	**1.1**
h-1-1-2-up*	28	16	3265	58		18		102		
simplified	23	13	5469	71	**0.8**	18	**1.0**	69	10	**1.3**
h-2-1-1-np*	43	26	831	20		192		201		
simplified	27	18	340	5	**3.8**	57	**3.4**	53	12	**3.1**
h-2-1-1-up*	47	29	2053	19		272		224		
simplified	29	19	489	6	**3.2**	57	**4.8**	64	12	**2.9**
h-2-1-2-np*	51	28	378805	6779		2816		415		
simplified	47	26	5477	71	**95.1**	498	**5.7**	252	11	**1.6**
h-2-1-2-up*	55	31	179133	3080		1378		417		
simplified	49	27	5477	71	**43.1**	138	**10.0**	261	11	**1.5**
h-2-2-1-np*	43	26	95637	1779		172		220		
simplified	24	17	1707	12	**147.0**	198	**0.9**	56	11	**3.3**
h-2-2-1-up*	47	29	1537	19		243		224		
simplified	24	17	888	8	**2.4**	111	**2.2**	63	14	**2.9**
h-2-2-2-np*	51	28	201009	3601		1631		407		
simplified	44	25	5469	71	**50.2**	273	**6.0**	218	11	**1.8**
h-2-2-2-up*	55	31	157786	2777		2238		400		
simplified	44	25	5477	71	**39.0**	317	**7.1**	219	12	**1.7**
h-3-1-1-np*	64	39	21850	199		1332		403		
simplified	40	27	3355	29	**6.8**	222	**6.0**	121	11	**3.0**
h-3-1-1-up*	69	43	15881	157		1798		487		
simplified	42	28	6800	47	**3.3**	222	**8.1**	138	15	**3.2**
h-3-1-2-np*	76	42	92752	1004		5458		769		
simplified	70	39	92752	1004	**1.0**	5458	**1.0**	518	12	**1.4**
h-3-1-2-up*	81	46	76307	932		5042		1006		
simplified	72	40	26489	195	**4.8**	818	**6.2**	549	12	**1.8**
h-3-2-1-np	64	39	21750	199		1552		404		
simplified	40	27	3327	29	**6.9**	204	**7.6**	122	12	**3.0**
h-3-2-1-up	69	43	18001	180		6069		489		
simplified	41	28	12726	78	**2.3**	968	**6.3**	133	12	**3.4**
h-3-2-2-np	76	42	92507	1004		7244		772		
simplified	70	39	92507	1004	**1.0**	7244	**1.0**	521	12	**1.4**
h-3-2-2-up	81	46	83473	1012		25908		1008		
simplified	71	40	21626	189	**5.4**	4092	**6.3**	573	16	**1.7**
h-3-3-1-np*	64	39	4182	44		745		404		
simplified	35	25	11607	67	**0.7**	461	**1.6**	94	12	**3.8**
h-3-3-1-up*	69	43	9316	110		4281		488		
simplified	35	25	6904	52	**2.1**	531	**8.1**	102	12	**4.3**
h-3-3-2-np*	76	42	26103	244		3279		771		
simplified	65	37	27894	232	**1.1**	2645	**1.2**	465	11	**1.6**
h-3-3-2-up*	81	46	60114	651		18258		1009		
simplified	65	37	30939	258	**2.5**	3431	**5.3**	465	13	**2.1**
h-4-1-1-np*	85	52	218389	2368		15467		782		
simplified	53	36	18478	166	**14.2**	717	**21.6**	179	12	**4.1**
h-4-1-1-up*	91	57	229290	2919		68608		1041		
simplified	55	37	36881	263	**11.1**	717	**95.7**	205	16	**4.7**
h-4-1-2-np*	101	56	1482857	21487		84025		2343		
simplified	93	52	1482857	21508	**1.0**	46378	**1.8**	997	12	**2.3**
h-4-1-2-up*	107	61	1256734	21699		220656		3236		
simplified	95	53	223639	1911	**11.4**	3878	**56.9**	1020	13	**3.1**
h-4-2-1-np	85	52	247426	2809		20098		787		
simplified	53	36	18450	166	**16.9**	699	**28.8**	180	16	**4.0**

h-4-2-1-up	91	57	333800	4257		116284		1065		
simplified	54	37	160813	1042	**4.1**	6661	**17.5**	213	13	**4.7**
h-4-2-2-np	101	56	1749092	26680		141532		2416		
simplified	93	52	1749092	26695	**1.0**	110566	**1.3**	1021	12	**2.3**
h-4-2-2-up	107	61	2557673	43937		839744		3599		
simplified	94	53	437426	4248	**10.3**	66876	**12.6**	1197	13	**3.0**
h-4-3-1-np	85	52	334620	3873		41213		831		
simplified	53	36	15460	143	**27.0**	598	**68.9**	180	14	**4.3**
h-4-3-1-up	91	57	312886	3961		102362		1103		
simplified	54	37	113912	1140	**3.5**	3719	**27.5**	211	15	**4.9**
h-4-3-2-np	101	56	2405020	37411		293300		2459		
simplified	93	52	2383829	37058	**1.0**	262898	**1.1**	1112	13	**2.2**
h-4-3-2-up	107	61	2485130	42706		798504		3575		
simplified	94	53	395887	3773	**11.3**	56830	**14.1**	1192	13	**3.0**
h-4-4-1-np*	85	52	29490	323		6819		778		
simplified	46	33	49051	351	**0.9**	2093	**3.3**	154	13	**4.7**
h-4-4-1-up*	91	57	161582	2196		75980		1042		
simplified	46	33	58771	417	**5.3**	3005	**25.3**	163	15	**5.9**
h-4-4-2-np*	101	56	213220	3340		47745		2296		
simplified	86	49	182899	1383	**2.4**	16340	**2.9**	950	13	**2.4**
h-4-4-2-up*	107	61	1070002	19979		441985		3293		
simplified	86	49	152799	1145	**17.4**	12366	**35.7**	945	13	**3.4**
h-5-1-1-np*	106	65	1950564	27563		235390		1611		
simplified	66	45	74143	737	**37.4**	2004	**117.5**	263	15	**5.8**
h-5-1-1-up*	113	71	4297963	74634		1415747		3297		
simplified	68	46	147716	1119	**66.6**	2004	**706.5**	294	15	**10.7**
h-5-1-2-np*	126	70						TO		
simplified	116	65	16311707	326493		324634		2366	14	
h-5-1-2-up*	133	76						TO		
simplified	118	66	1343179	13239		15506		1823	14	
h-5-5-1-np*	106	65	401718	3804		46346		1509		
simplified	57	41	343891	2548	**1.5**	5882	**7.9**	253	17	**5.6**
h-5-5-1-up*	113	71	2324276	37556		933524		2697		
simplified	57	41	258355	1875	**20.0**	4640	**201.2**	252	14	**10.1**
h-5-5-2-np*	126	70	4264578	58404		648894		7059		
simplified	107	61	458365	4561	**12.8**	35639	**18.2**	1933	13	**3.6**
h-5-5-2-up*	133	76						TO		
simplified	107	61	601267	6013		63101		1957	14	
h-6-1-1-np*	127	78						TO		
simplified	79	54	241750	2607		5007		372	15	
h-6-1-1-up*	135	85						TO		
simplified	81	55	481643	3905		5007		418	16	
h-6-1-2-np*	151	84						TO		
simplified	139	78						TO	16	
h-6-1-2-up*	159	91						MO		
simplified	141	79	6391567	72893		54266		3314	17	
h-6-6-1-np*	127	78	3097533	32860		644092		3198		
simplified	68	49	1832644	14540	**2.3**	10167	**63.4**	471	14	**6.6**
h-6-6-1-up*	135	85						TO		
simplified	68	49	2230219	17081		13316		492	15	
h-6-6-2-np*	151	84						TO		
simplified	128	73	6818683	96757		326943		5361	14	
h-6-6-2-up*	159	91						MO		
simplified	128	73	4710191	57250		400970		4867	15	
h-7-1-1-np	148	91						TO		
simplified	92	63	678967	7932		11442		569	17	
h-7-1-1-up	157	99						TO		
simplified	94	64	1353074	11787		11442		660	16	
h-7-1-2-np	176	98						MO		
simplified	162	91						TO	16	
h-7-1-2-up	185	106						MO		
simplified	164	92	25612855	338843		170546		6534	18	
h-7-7-1-np	148	91						TO		
simplified	79	57	5035735	41776		27898		1058	15	
h-7-7-1-up	157	99						TO		
simplified	79	57	3957139	47334		45452		1033	18	
h-7-7-2-np	176	98						MO		
simplified	149	85						TO	14	
h-7-7-2-up	185	106						MO		
simplified	149	85						TO	15	
h-8-1-1-np	169	104						TO		
simplified	105	72	1702132	21516		24312		850	16	

h-8-1-1-up	179	113						TO		
simplified	107	73	3392969	33615		24312		1077	18	
h-8-1-2-np	201	112						MO		
simplified	185	104						MO	15	
h-8-1-2-up	211	121						MO		
simplified	187	105						TO	16	
h-8-8-1-np	169	104						TO		
simplified	90	65	8694964	71121		103105		2003	17	
h-8-8-1-up	179	113						TO		
simplified	90	65	9702307	74744		29575		2086	19	
h-8-8-2-np	201	112						MO		
simplified	170	97						TO	15	
h-8-8-2-up	211	121						MO		
simplified	170	97						TO	20	
CM										
simplified	108	64						TO	12	
c-1-1-2-dp*	23	12	10682	91		28		83		
simplified	21	12	297	6	**15.2**	28	**1.0**	68	10	**1.1**
c-1-1-2-sp*	21	11	6438	56		18		75		
simplified	18	10	1042	13	**4.3**	18	**1.0**	61	11	**1.0**
c-1-1-3-dp*	31	16	64134	919		46		140		
simplified	28	16	6546	54	**17.0**	46	**1.0**	93	11	**1.3**
c-1-1-3-sp*	29	15	78662	1132		28		132		
simplified	25	14	6357	53	**21.4**	28	**1.0**	85	10	**1.4**
c-1-1-4-dp*	40	21	61924	890		140		193		
simplified	36	21	5697	33	**27.0**	140	**1.0**	132	12	**1.3**
c-1-1-4-sp*	38	20	132893	1552		81		174		
simplified	33	19	5697	33	**47.0**	81	**1.0**	122	10	**1.3**
c-1-1-5-dp*	48	25	1279080	34019		382		650		
simplified	43	25	69997	559	**60.9**	418	**0.9**	175	12	**3.5**
c-1-1-5-sp*	46	24	1140848	15420		213		281		
simplified	40	23	70237	563	**27.4**	231	**0.9**	164	10	**1.6**
c-1-1-6-dp*	57	30	15107242	442479		835		4709		
simplified	51	30	19029	135	**3277.6**	506	**1.7**	224	11	**20.1**
c-1-1-6-sp*	55	29	933104	14506		368		353		
simplified	48	28	19029	135	**107.5**	279	**1.3**	211	11	**1.6**
c-1-1-7-dp*	65	34						MO		
simplified	58	34	486969	2760		2816		288	11	
c-1-1-7-sp*	63	33	4329730	58326		1332		835		
simplified	55	32	484857	2728	**21.4**	1514	**0.9**	271	13	**2.9**
c-1-1-8-dp*	74	39						MO		
simplified	66	39	588461	3663		1485		363	12	
c-1-1-8-sp*	72	38	21137311	399353		8220		2808		
simplified	63	37	588461	3661	**109.1**	796	**10.3**	350	10	**7.8**
c-1-1-9-dp*	82	43						TO		
simplified	73	43	329404	3113		27196		467	12	
c-1-1-9-sp*	80	42						TO		
simplified	70	41	657362	6226		10983		452	14	
c-1-1-10-dp*	91	48						TO		
simplified	81	48	2814489	27695		43154		663	11	
c-1-1-10-sp*	89	47						TO		
simplified	78	46	2825753	27889		22598		648	14	
c-1-1-11-dp*	99	52						TO		
simplified	88	52	9303636	67587		150570		1075	12	
c-1-1-11-sp*	97	51						TO		
simplified	85	50	9303636	67586		78206		1005	11	
c-1-1-12-dp*	108	57						TO		
simplified	96	57	23908239	192709		249672		2364	12	
c-1-1-12-sp*	106	56						TO		
simplified	93	55	13187345	180848		4		1497	12	
c-1-1-13-sp*	114	60						TO		
simplified	100	59	44683774	490519		1111109		6310	12	
c-1-1-14-sp*	123	65						TO		
c-2-1-2-dp	47	24	359352	3328		380		258		
simplified	43	24	1477	15	**221.9**	253	**1.5**	179	11	**1.4**
c-2-1-2-sp	43	22	113123	1040		131		226		
simplified	37	20	1022	13	**80.0**	88	**1.5**	158	11	**1.3**
c-2-1-3-dp	63	32	4434806	52692		1428		709		
simplified	57	32	533064	10534	**5.0**	740	**1.9**	306	11	**2.2**
c-2-1-3-sp	59	30	2338496	28618		431		562		
simplified	51	28	870532	14131	**2.0**	397	**1.1**	310	11	**1.7**

c-2-1-4-dp	81	42	1042324	20791		7414		894		
simplified	73	42	167252	1652	**12.6**	8769	**0.8**	404	11	**2.2**
c-2-1-4-sp	77	40	131727	1533		2079		575		
simplified	67	38	60687	544	**2.8**	2607	**0.8**	359	12	**1.6**
c-2-2-2-dp	47	24	543036	8704		371		269		
simplified	40	23	1116	12	**725.3**	185	**2.0**	157	11	**1.6**
c-2-2-2-sp	43	22	324955	5171		117		231		
simplified	34	19	456	6	**861.8**	77	**1.5**	133	13	**1.6**
c-2-2-3-dp	63	32	3571206	26363		9451		634		
simplified	54	31	6357	53	**497.4**	548	**17.2**	239	11	**2.5**
c-2-2-3-sp	59	30	1986404	15369		487		496		
simplified	48	27	6546	54	**284.6**	208	**2.3**	207	14	**2.2**
c-2-2-4-dp	81	42	1197420	21574		10286		928		
simplified	70	41	58575	659	**32.7**	6651	**1.5**	372	12	**2.4**
c-2-2-4-sp	77	40	131727	1533		2054		576		
simplified	64	37	14692	194	**7.9**	2547	**0.8**	322	12	**1.7**
c-3-1-2-dp*	70	36	29767	332		1828		441		
simplified	64	36	29767	332	**1.0**	1828	**1.0**	326	11	**1.3**
c-3-1-2-sp*	64	33	6344	56		343		384		
simplified	55	30	4852	47	**1.2**	343	**1.0**	283	11	**1.3**
c-3-1-3-dp*	94	48						TO		
simplified	85	48	157791	2012		9260		545	12	
c-3-1-3-sp*	88	45	21067228	165210		5011		3027		
simplified	76	42	22872	248	**666.2**	1451	**3.5**	465	12	**6.3**
c-3-1-4-dp*	121	63	8671123	205172		310132		2166		
simplified	109	63	11465000	196120	**1.0**	797950	**0.4**	1847	15	**1.2**
c-3-1-4-sp*	115	60	1707356	24957		82413		1487		
simplified	100	57	2096338	28742	**0.9**	98569	**0.8**	858	15	**1.7**
c-3-2-2-dp*	70	36	29678	332		2991		443		
simplified	64	36	29678	332	**1.0**	2991	**1.0**	327	14	**1.3**
c-3-2-2-sp*	64	33	6344	56		332		385		
simplified	55	30	4830	47	**1.2**	332	**1.0**	284	12	**1.3**
c-3-2-3-dp*	94	48						TO		
simplified	85	48	157562	2010		15863		546	12	
c-3-2-3-sp*	88	45	20855116	163589		4978		3086		
simplified	76	42	22834	248	**659.6**	1432	**3.5**	465	12	**6.5**
c-3-2-4-dp*	121	63	8745830	207517		703963		2373		
simplified	109	63	17402416	298823	**0.7**	984545	**0.7**	2414	12	**1.0**
c-3-2-4-sp*	115	60	1709142	24175		84922		1543		
simplified	100	57	2064924	27077	**0.9**	128294	**0.7**	864	12	**1.8**
c-3-3-2-dp*	70	36	10521	92		2935		473		
simplified	59	34	15634	137	**0.7**	2170	**1.4**	284	13	**1.6**
c-3-3-2-sp*	64	33	6344	56		546		411		
simplified	50	28	3321	30	**1.9**	553	**1.0**	227	14	**1.7**
c-3-3-3-dp*	94	48	10759574	117530		24181		1625		
simplified	80	46	53460	689	**170.6**	11828	**2.0**	494	15	**3.2**
c-3-3-3-sp*	88	45	6048731	67634		3754		1386		
simplified	71	40	14130	166	**407.4**	2888	**1.3**	393	12	**3.4**
c-3-3-4-dp*	121	63	5917902	128447		720129		2363		
simplified	104	61	4670873	83248	**1.5**	473260	**1.5**	1184	12	**2.0**
c-3-3-4-sp*	115	60	769756	11080		112343		1444		
simplified	95	55	825088	17702	**0.6**	122436	**0.9**	723	12	**2.0**
RC										
rc-01-01-hc*	35	21	47400	1034		328		170		
simplified	19	13	207	5	**198.9**	48	**6.8**	54	11	**2.6**
rc-01-01-sc*	15	9	452	8		8		54		
simplified	8	5	106	4	**2.1**	7	**1.1**	16	9	**2.1**
rc-01-02-hc	42	24	68470	1494		571		222		
simplified	22	15	618	7	**219.7**	111	**5.1**	55	11	**3.4**
rc-01-02-sc	22	12	790	12		11		92		
simplified	11	7	172	4	**2.8**	9	**1.2**	26	11	**2.5**
rc-01-03-hc	48	27	89540	1954		687		271		
simplified	25	17	708	8	**241.2**	129	**5.3**	64	11	**3.6**
rc-01-03-sc	28	15	980	14		14		118		
simplified	14	9	184	5	**3.0**	11	**1.3**	33	10	**2.8**
rc-01-04-hc	54	30	17929	396		387		321		
simplified	28	19	2436	16	**25.1**	219	**1.8**	82	11	**3.4**
rc-01-04-sc	34	18	314	6		17		162		
simplified	17	11	199	5	**1.1**	13	**1.3**	49	11	**2.7**
rc-01-05-hc*	60	33	17929	394		448		381		
simplified	31	21	2482	21	**18.9**	282	**1.6**	81	11	**4.1**

rc-01-05-sc*	40	21	324	6		20		202		
simplified	20	13	493	7	**0.9**	15	**1.3**	47	10	**3.5**
rc-01-10-hc*	90	48	17929	394		891		890		
simplified	46	31	17724	97	**4.1**	926	**1.0**	191	12	**4.4**
rc-01-10-sc*	70	36	1124	18		35		537		
simplified	35	23	1177	18	**1.0**	25	**1.4**	119	11	**4.1**
rc-01-15-hc*	120	63	89034	1355		1727		2201		
simplified	61	41	15680	145	**9.4**	723	**2.4**	368	12	**5.8**
rc-01-15-sc*	100	51	4904	131		50		1379		
simplified	50	33	3437	55	**2.4**	35	**1.4**	228	12	**5.7**
rc-01-20-hc*	150	78	189166	3783		1982		5332		
simplified	76	51	49523	368	**10.3**	3801	**0.5**	759	14	**6.9**
rc-01-20-sc*	130	66	12232	327		65		3575		
simplified	65	43	3135	79	**4.1**	45	**1.4**	460	12	**7.6**
rc-01-25-hc*	180	93						TO		
simplified	91	61	37848	293		1879		1605	15	
rc-01-25-sc*	160	81						TO		
simplified	80	53	8110	244		55		990	13	
rc-01-30-hc*	210	108						TO		
simplified	106	71	29315	1172		777		3306	18	
rc-01-30-sc*	190	96						TO		
simplified	95	63	11166	385		65		2054	13	
rc-02-01-hc	43	26	68424	1493		526		225		
simplified	22	15	348	6	**257.4**	57	**9.2**	55	11	**3.4**
rc-02-01-sc	23	14	1598	27		10		110		
simplified	11	7	172	4	**6.4**	9	**1.1**	26	10	**3.0**
rc-02-02-hc	50	29	100006	2222		749		294		
simplified	25	17	483	7	**317.4**	84	**8.9**	64	11	**3.9**
rc-02-02-sc	30	17	2204	36		14		150		
simplified	14	9	184	5	**7.8**	12	**1.2**	33	10	**3.5**
rc-02-03-hc	56	32	131588	2882		885		358		
simplified	28	19	3060	19	**150.1**	280	**3.2**	74	13	**4.1**
rc-02-03-sc	36	20	3725	60		18		188		
simplified	17	11	464	5	**11.1**	15	**1.2**	40	12	**3.6**
rc-02-04-hc	62	35	831	20		277		402		
simplified	31	21	2483	21	**0.9**	327	**0.8**	100	12	**3.6**
rc-02-04-sc	42	23	1745	32		22		229		
simplified	20	13	592	7	**4.4**	18	**1.2**	60	11	**3.2**
rc-03-01-hc	50	31	1388	20		152		282		
simplified	25	17	440	7	**2.8**	75	**2.0**	71	14	**3.3**
rc-03-01-sc	30	19	2348	36		12		148		
simplified	14	9	186	5	**7.6**	11	**1.1**	33	10	**3.5**
rc-03-02-hc	57	34	831	20		221		356		
simplified	28	19	2748	18	**1.1**	255	**0.9**	74	12	**4.1**
rc-03-02-sc	37	22	3110	50		17		194		
simplified	17	11	384	5	**9.5**	15	**1.1**	41	11	**3.8**
rc-03-03-hc	63	37	831	20		294		428		
simplified	31	21	1503	14	**1.4**	147	**2.0**	91	11	**4.2**
rc-03-03-sc	43	25	4570	63		22		235		
simplified	20	13	593	7	**8.7**	19	**1.2**	53	11	**3.7**
rc-03-04-hc	69	40	898	20		352		515		
simplified	34	23	3678	26	**0.8**	597	**0.6**	116	12	**4.0**
rc-03-04-sc	49	28	315	6		27		282		
simplified	23	15	681	9	**0.6**	23	**1.2**	69	11	**3.5**
rc-04-01-hc	57	36	831	20		139		370		
simplified	28	19	2748	18	**1.1**	165	**0.8**	82	11	**4.0**
rc-04-01-sc	37	24	315	6		14		182		
simplified	17	11	384	5	**1.1**	13	**1.1**	40	11	**3.6**
rc-04-02-hc	64	39	831	20		225		464		
simplified	31	21	3042	25	**0.8**	498	**0.5**	92	12	**4.5**
rc-04-02-sc	44	27	315	6		20		237		
simplified	20	13	593	7	**0.8**	18	**1.1**	53	11	**3.7**
rc-04-03-hc	70	42	831	20		291		554		
simplified	34	23	1878	18	**1.1**	354	**0.8**	117	12	**4.3**
rc-04-03-sc	50	30	315	6		26		293		
simplified	23	15	770	10	**0.6**	23	**1.1**	61	11	**4.0**
rc-04-04-hc	76	45	1114	20		447		667		
simplified	37	25	12576	71	**0.3**	1104	**0.4**	137	12	**4.5**
rc-04-04-sc	56	33	334	7		32		368		
simplified	26	17	1352	12	**0.6**	28	**1.1**	76	14	**4.1**
rc-05-01-hc*	64	41	3228	28		166		510		
simplified	31	21	1643	15	**1.9**	219	**0.8**	101	12	**4.5**

rc-05-01-sc*	44	29	315	6		16		250		
simplified	20	13	493	7	**0.9**	15	**1.1**	46	11	**4.4**
rc-05-05-hc*	89	53	1552	21		656		1036		
simplified	43	29	11958	79	**0.3**	1119	**0.6**	173	13	**5.6**
rc-05-05-sc*	69	41	652	10		44		577		
simplified	32	21	1890	20	**0.5**	39	**1.1**	107	12	**4.9**
rc-10-01-hc*	99	66	17514	244		432		2712		
simplified	46	31	13668	76	**3.2**	475	**0.9**	175	13	**14.4**
rc-10-01-sc*	79	54	1514	26		26		1357		
simplified	35	23	1819	23	**1.1**	25	**1.0**	119	13	**10.3**
rc-10-10-hc*	154	93						TO		
simplified	73	49	108984	1052		1805		673	16	
rc-10-10-sc*	134	81	11062	150		134		5636		
simplified	62	41	9819	158	**0.9**	124	**1.1**	403	17	**13.4**
rc-15-01-hc*	134	91						TO		
simplified	61	41	4578	93		363		355	15	
rc-15-01-sc*	114	79						TO		
simplified	50	33	3437	65		35		221	14	
rc-15-15-hc*	219	133						TO		
simplified	103	69	148217	1960		2469		2945	18	
rc-15-15-sc*	199	121						TO		
simplified	92	61	29055	681		259		1802	17	
rc-20-01-hc*	169	116						TO		
simplified	76	51	13682	110		435		752	16	
rc-20-01-sc*	149	104						TO		
simplified	65	43	4281	77		45		452	16	
rc-20-20-hc*	284	173						TO		
simplified	133	89						TO	24	
rc-20-20-sc*	264	161						TO		
simplified	122	81	63011	1984		444		7145	22	
rc-25-01-hc*	204	141						TO		
simplified	91	61	27810	938		583		1597	18	
rc-25-01-sc*	184	129						TO		
simplified	80	53	4164	137		55		956	17	
rc-25-25-hc*	349	213						TO		
simplified	163	109						MO	27	
rc-25-25-sc*	329	201						TO		
simplified	152	101						MO	26	
rc-30-01-hc*	239	166						TO		
simplified	106	71	12547	507		579		3291	21	
rc-30-01-sc*	219	154						TO		
simplified	95	63	10664	231		65		2031	18	
rc-30-30-hc*	414	253						TO		
simplified	193	129						MO	38	
rc-30-30-sc*	394	241						TO		
simplified	182	121						MO	36	
SF										
sf-01-02*	13	7	205	4		18		46		
simplified	11	7	177	4	**1.0**	16	**1.1**	35	8	**1.1**
sf-01-04*	17	9	3533	34		30		60		
simplified	13	9	1482	20	**1.7**	22	**1.4**	38	9	**1.3**
sf-01-06*	21	11	15012	167		42		82		
simplified	15	11	2070	24	**7.0**	28	**1.5**	50	9	**1.4**
sf-01-08*	25	13	363360	6274		54		166		
simplified	17	13	2964	41	**152.3**	34	**1.6**	58	10	**2.5**
sf-01-10*	29	15	4575848	85478		66		665		
simplified	19	15	6850	85	**1003.3**	40	**1.7**	66	10	**8.8**
sf-01-12*	33	17	5996166	129484		78		706		
simplified	21	17	9386	148	**874.9**	46	**1.7**	84	12	**7.4**
sf-01-14*	37	19						TO		
simplified	23	19	11236	206		52		92	11	
sf-01-16*	41	21						MO		
simplified	25	21	13346	283		58		110	13	
sf-01-18*	45	23						MO		
simplified	27	23	16702	379		64		120	11	
sf-02-02*	27	14	546	8		75		119		
simplified	20	13	902	13	**0.6**	63	**1.2**	73	11	**1.4**
sf-02-04*	35	18	3533	34		243		159		
simplified	24	17	994	11	**3.1**	129	**1.9**	93	10	**1.5**
sf-02-06*	43	22	15012	167		507		228		
simplified	28	21	37591	404	**0.4**	219	**2.3**	127	11	**1.7**

sf-02-08*	51	26	363360	6305		867		404		
simplified	32	25	1530202	39666	**0.2**	333	**2.6**	384	11	**1.0**
sf-02-10*	59	30	4559930	84839		1323		1110		
simplified	36	29	245790	6078	**14.0**	471	**2.8**	178	11	**5.9**
sf-02-12*	67	34	5996166	129405		1875		1655		
simplified	40	33	359896	10347	**12.5**	633	**3.0**	256	12	**6.2**
sf-02-14*	75	38						TO		
simplified	44	37	254262	4829		819		309	12	
sf-02-16*	83	42						MO		
simplified	48	41	368304	8393		1029		449	11	
sf-02-18*	91	46						MO		
simplified	52	45	513782	12201		1263		654	11	
sf-03-02*	40	21	2370	27		223		202		
simplified	29	19	8276	119	**0.2**	183	**1.2**	115	10	**1.6**
sf-03-04*	52	27	20301	199		1275		295		
simplified	35	25	11409	68	**2.9**	521	**2.4**	152	11	**1.8**
sf-03-06*	64	33	60681	690		3799		457		
simplified	41	31	23829	144	**4.8**	1131	**3.4**	174	11	**2.5**
sf-04-02*	53	28	10641	95		528		307		
simplified	38	25	9553	197	**0.5**	428	**1.2**	183	12	**1.6**
sf-04-04*	69	36	106989	1199		4953		490		
simplified	46	33	32639	579	**2.1**	1613	**3.1**	245	12	**1.9**
sf-04-06*	85	44	455721	6488		20478		841		
simplified	54	41	101034	1548	**4.2**	4353	**4.7**	313	12	**2.6**
sf-05-02*	66	35	25281	232		1074		444		
simplified	47	31	29469	412	**0.6**	864	**1.2**	271	11	**1.6**
sf-05-04*	86	45	415941	5321		15645		782		
simplified	57	41	136149	1971	**2.7**	4161	**3.8**	388	12	**2.0**
sf-05-06*	106	55	2457201	43147		87636		1526		
simplified	67	51	435669	6753	**6.4**	13665	**6.4**	552	12	**2.7**
sf-06-02*	79	42	51489	486		1963		621		
simplified	56	37	196729	3761	**0.1**	1571	**1.2**	412	12	**1.5**
sf-06-04*	103	54	876066	16618		42507		1222		
simplified	68	49	1335824	26021	**0.6**	9411	**4.5**	670	12	**1.8**
sf-06-06*	127	66	7010802	202277		315955		2842		
simplified	80	61	1306869	11628	**17.4**	36963	**8.5**	976	13	**2.9**
sf-07-02*	92	49	112126	1519		3315		854		
simplified	65	43	222482	5650	**0.3**	2643	**1.3**	652	11	**1.3**
sf-07-04*	120	63	2380338	50393		102963		1910		
simplified	79	57	1796599	43956	**1.1**	19275	**5.3**	1147	13	**1.6**
sf-07-06*	148	77	25276322	845946		997155		6751		
simplified	93	71	7784118	183272	**4.6**	89235	**11.2**	2330	13	**2.9**
sf-08-02*	105	56	671117	8582		5268		1170		
simplified	74	49	428050	8654	**1.0**	4188	**1.3**	1031	15	**1.1**
sf-08-04*	137	72	8648781	147752		227802		3115		
simplified	90	65	3336275	60763	**2.4**	36534	**6.2**	1981	13	**1.6**
sf-08-06*	169	88						TO		
simplified	106	81	16679889	267579		196914		4753	14	
sf-09-02*	118	63	805218	11320		7978		1601		
simplified	83	55	1076857	27247	**0.4**	6328	**1.3**	1709	13	**0.9**
sf-09-04*	154	81	19135257	352555		468328		5283		
simplified	101	73	15760349	401191	**0.9**	65068	**7.2**	4315	15	**1.2**
sf-09-06*	190	99						TO		
simplified	119	91						TO	16	
MOVARES										
cat2-spare1	120	102						TO		
simplified	19	12	282	6		14		64	9	
cat2-spare2	126	105						TO		
simplified	27	17	802	8		38		101	9	
cat2-spare3	114	99						TO		
simplified	10	6	73	3		8		34	9	
fa-rk-bb-sh-spare	118	88	416976	10720		16655		1900		
simplified	96	68	416976	10498	**1.0**	16655	**1**	1122	8	**1.7**
fa-rk-bb-spare	118	88	14596	128		2527		1479		
simplified	96	68	14596	128	**1.0**	2527	**1**	858	8	**1.7**
ge-aut-sh-spare	113	93						MO		
simplified	32	17						MO	9	
ge-aut-spare	125	105	1832	36		32		4752		
simplified	44	29	2224	36	**1.0**	32	**1**	192	9	**23.6**
to-act-ovw-spare	390	367						MO		
simplified	32	21	1889	19		93		104	248	

HCAS										
cas-hecs	41	25	12928	221		122		189		
simplified	26	15	860	16	**13.6**	34	**3.6**	97	7	**1.8**
cas	16	11	89	3		6		44		
simplified	5	3	33	3	**1.1**	6	**1.0**	10	6	**2.8**
hcas-M1o3-PMPP	37	18	12557	165		56		180		
simplified	34	17	424	6	**26.8**	158	**0.4**	154	7	**1.1**
hcas-M1o3	33	16	190	4		56		145		
simplified	28	15	190	4	**1.0**	56	**1.0**	109	6	**1.3**
hcas-M2o3-PMPP	37	18	9937	131		46		181		
simplified	34	17	412	6	**20.9**	160	**0.3**	144	7	**1.2**
hcas-M2o3	33	16	180	4		46		145		
simplified	28	15	180	4	**1.0**	46	**1.0**	109	6	**1.3**
hcas-M3o3-PMPP	37	18	6007	80		28		181		
simplified	30	17	1603	21	**3.8**	70	**0.4**	133	7	**1.3**
hcas-M3o3	33	16	214	4		28		140		
simplified	24	15	1492	19	**0.2**	28	**1.0**	101	6	**1.3**
hcas	20	10	84	3		16		72		
simplified	14	9	108	3	**1.0**	16	**1.0**	47	6	**1.4**
SAP										
sap-sc00	22	12	1796	26		26		83		
simplified	10	6	132	4	**6.8**	26	**1.0**	33	10	**1.9**
sap-sc01	22	12	704	12		8		95		
simplified	7	4	55	3	**3.6**	8	**1.0**	15	10	**3.9**
sap-sc10	22	12	72	4		1		95		
simplified	22	12	33	3	**1.3**	1	**1.0**	14	10	**4.0**
sap-sc11	22	12	252	6		3		87		
simplified	1	1	4	3	**2.0**	4	**0.8**	5	10	**6.0**

B. Detailed environment information

.

All experiments were executed on a PC with the following key characteristics

- CPU: `Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz (L2 cache: 256K L3 cache: 8192K)`
- RAM: `8GB DDR3 @ 1333MHz`
- OS: `Debian GNU/Linux 8.0 (jessie) Kernel(Linux 3.16.0-4)`

We used the following software and versions:

- With regard to DFTCalc:
  c++-compiler `gcc 4.9.1`
  CADP `VERSION 2014-j "Amsterdam"`
  DFTCalc `commit fc21b7c006`
  MRMC `version 1.5`
  as well as some other packages in their latest stable release (as of 01-01-2015).
- With regard to Groove, GroovyDFT and aDFTpub:
  Java `OpenJDK Runtime Environment (IcedTea 2.5.3) (7u71-2.5.3-2)`
  Groove `version 5.4.0 precompiled.`
  Scala`version 2.10`
  sbt[Scala build tool] `verion 0.13-7`
  Python 3`version 3.4.2`

No other software is necessary to execute all experiments described in the paper. The developed tools are available for download via `http://moves.rwth-aachen.de/ft-diet/`.

List of Symbols

General

$\mathbb{1}_s$ The Dirac distribution with support $\{s\}$. 8

$\text{Distr}(S)$ The set of all distributions over S. 8

$\mathbb{E}(X)$ The expected value X evaluates to. 8

μ discrete probability distribution. 8

$\mathcal{P}(K)$ Power set of K. 7

$u(x)$ The heavyside function. 7

$\mathbf{0}$ The zero function. 7

Markov Automata

$\approx_s$ Weak bisimimulation. 16

$\approx_w$ Weak bisimimulation. 16

$\Pr^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond^{\leq t} G)$ Time-bounded reachability probability to reach G from s within t. 14

$\mathsf{ET}^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond G)$ Expected time to reach G from s within t. 15

$\Pr^{\mathcal{M}}_{\mathcal{S}}(s, \Diamond G)$ Unbounded reachability probability to reach G from s. 14

DFTs

$\text{Active}(\pi)$ Active elements after occurence of π. 95

$\text{Activated}(\pi)$ Active components after occurence of π. 95

α_ω Dormancy factor of component ω. 74

$\text{Available}(\pi, v)$ Set of elements that can be claimed by v after π. 78

$\text{ClaimedBy}(\pi, v)$ Set of spare-gates which claimed π after v. 77

Ω Set of component failures. 73

$\mathcal{C}_F$ Computation tree of F. 96

$\mathcal{C}^*_F$ Computation tree of F with only top as label.. 96

$\mathfrak{C}(s)$ Configuration of a state from the computation tree. 96

$\Delta(\pi)$ Events which are triggered after π. 92

$E(\sigma)$ Edges implicetely defined by a successor function σ. 73

$\prec$ element hierarchy relation. 73

EM_r Extended module represented by r. 74

$\text{Failable}(\pi)$ Set of PANDS which can still fail after π. 84

$\text{Failed}(\pi)$ Set of failed elements after occurence of π. 77

$\text{FB}_\pi(x, y)$ Statement that y did not fail after x at π. 78

$\text{Failed}[f](\pi)$ Set of failed elements after occurence of π, considering an oracle f. 133

$\mathcal{R}(\omega)$ Failure rate of component ω. 74

$\text{ifcp}(x)$ Set of paths along which a failure can propage to x. 89

$\text{JustFailed}(\pi)$ Set of elements which failed after π, but not earlier. 84

$\text{LastClaimed}(\pi, v)$ Last claimed element by v. 87

$\bowtie$ element module relation. 74

MTTF_F MTTF of F. 100

π Event trace. 75

Pol Policy on a DFT. 97

$\mathsf{Pol}_\emptyset$ Policy on a DFT without any restrictions. 97

$\theta(s)$ Predecessor of an element s. 73

$\theta^*(s)$ Predecessor closure of an element s. 73

PrF_F Probability of failure of F. 100

$w_\downarrow$ Last element of word w. 75

$\mathrm{pre}(w)$ Set of prefixes of w. 75

$\mathrm{Rely}(t)_F$ Reliability of F with mission time t. 100

ρ Failure trace. 75

$\Sigma^{\triangleright}$ Word over Σ without repetitions. 75

$\sigma^*(v)$ successor closure of an element v. 73

$\mathrm{spmp}(a, b)$ Set of paths from a to b which do not cross modules. 73

$\equiv$ strong equivalence of DFTs. 101

$\mathcal{T}_F$ Functional transducer of F. 92

F_X Elements in DFT F of type X. 73

$\approxeq$ weak equivalence of DFTs. 102

Rewriting

$\mathfrak{C}$ Context restriction of a DFT rewrite rule. 119

η Mapping from result subDFT to result DFT. 122

h_r Mapping from $V_\mathfrak{i} \cup V_\mathfrak{o}$ to R. 119

$\mathrm{InV}(v)$ Vertices connected by an incoming edge to v. 18

κ Mapping from L to host DFT. 121

L Left-hand side of a DFT rewrite rule. 119

$\mathrm{le}(e)$ edge labels for e. 18

$\mathrm{lv}(v)$ vertex labels for v. 18

ν Mapping from $\kappa(V_\mathfrak{i} \cup V_\mathfrak{o})$ to host DFT. 122

ν' Mapping from $\kappa(V_\mathfrak{i} \cup V_\mathfrak{o})$ to result DFT. 122

$\mathrm{OutV}(v)$ Vertices connected by an outgoing edge from v. 18

R Right-hand side of a DFT rewrite rule. 119

$\mathfrak{r}$ DFT rewrite rule. 120

$V_\mathfrak{i}$ Input interface of a DFT rewrite rule. 119

$V_\mathfrak{o}$ Output interface of a DFT rewrite rule. 119

Index