DOI 10.1007/s00165-016-0412-0
BCS © 2017
Formal Aspects of Computing

Formal Aspects of Computing

Fault trees on a diet: automated reduction by graph rewriting

Sebastian Junges[1], Dennis Guck[2], Joost-Pieter Katoen[1,2], Arend Rensink[2] and Mariëlle Stoelinga[2]

[1] Software Modeling and Verification, RWTH Aachen University, Aachen, Germany
[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands

Abstract. Fault trees are a popular industrial technique for reliability modelling and analysis. Their extension with common reliability patterns, such as spare management, functional dependencies, and sequencing—known as *dynamic* fault trees (DFTs)—has an adverse effect on scalability, prohibiting the analysis of complex, industrial cases. This paper presents a novel, fully automated reduction technique for DFTs. The key idea is to interpret DFTs as directed graphs and exploit graph rewriting to simplify them. We present a collection of rewrite rules, address their correctness, and give a simple heuristic to determine the order of rewriting. Experiments on a large set of benchmarks show substantial DFT simplifications, yielding state space reductions and timing gains of up to two orders of magnitude.

Keywords: Fault tree analysis, Dynamic fault trees, Reliability, Graph rewriting.

1. Introduction

Since the 1970s, probabilistic safety assessment is a well-established technique to numerically quantify risk measures in safety-critical systems. It aims at determining the scenarios that lead to a system failure, assesses their likelihood, and yields information about the causes of these system failures. Typical measures of interest are the system reliability (i.e., what is the probability that the system is operational up to time t?) and availability (i.e., what is the expected up time?). Fault tree analysis [SVD+02] is one of the most prominent safety assessment technique. It is standardised by the IEC [IEC07], and deployed by many companies and institutions, like FAA, NASA, ESA, Airbus, Honeywell, etc. Fault trees (FTs) model the various parallel and sequential combinations of failures of the various system components that ultimately may lead to an undesired event (e.g., core damage); for a recent survey see [RS15]. FTs model how failures propagate through the system: the leaves model individual component failures or human errors and are equipped with continuous probability distributions describing the random failure behaviour of the component. The internal nodes of an FT—referred to as gates—model how component failures lead to system failures.

Correspondence and offprint requests to: S. Junges, E-mail: sebastian.junges@cs.rwth-aachen.de

Published online: 20 January 2017

Static fault trees. Typical gates in static FTs are logical gates such as AND and OR-gates. Due to, e.g., redundancy, not every single component failure leads to a system failure. The probability distributions of the leaves are propagated in a bottom-up fashion through the tree according to the logic of the gates to reach a probability distribution of the top event, the root of the FT. The analysis of static FTs is rather straightforward as only logical gates are used (without negation), and thus gates have no internal state. Most analysis techniques for static FTs are based on determining the minimal cut sets (MCSs), the set of minimal failure combinations that lead to the top event failure. This can be efficiently done by succinct data structures such as binary decision diagrams (BDDs). The probability calculations are then carried out on top of this logical analysis using the MCSs.

Dynamic fault trees. *Dynamic fault trees* (DFTs) [DBB92, SVD+02] are directed acyclic graphs that are more expressive and more involved than static fault trees. They cater for common dependability patterns, such as spare management, functional dependencies, and sequencing, see also [JGKS16], and have in common that their behaviour is dynamic, that is, their behaviour not only depends on the set of failed components, but possibly also their order. Besides the logical gates as in static FT, DFTs feature gates that have more involved behaviour. For instance, the PAND-gate fails if all its children have failed in a left-to-right order; if this ordering is violated, the PAND does never fail; it is so-called fail-safe. The treatment of spare components (such as spare tires in a car) is modelled by a SPARE-gate. If its child, the component at hand, fails, it can be replaced by its spare; if that spare also fails, and there are no other spares left, the SPARE gate fails. Though this behaviour seems relatively simple at first sight, the fact that spare components may be shared by various SPARE gates, and may themselves be DFTs (and not just leaves), complicates matters considerably. It is commonly assumed that component failures are governed by negative exponential distributions. Though this is sometimes a simplification, this abstraction is well-suited in cases only mean values of failure rates are known. (Technically speaking, the distribution that maximises the entropy whenever only the mean value $1/\lambda$ is known is the exponential distribution with rate λ.) Thus, the DFT behaviour is described by continuous-time Markov chains (CTMCs), where transitions correspond to the failure of a basic event.

The analysis of DFTs relies on extracting an underlying stochastic model, such as a Bayesian network [BPMC01, BD06], a (variant of a) CTMC [DVG97, BCS10] or a stochastic Petri net [Rai05, BFGP03] whose semantics is a CTMC too. The state-of-the-art techniques rely on generating CTMCs. The *state space generation* process—how to obtain a representation of the CTMC—is one of the main bottlenecks in DFT analysis. In fact, the analysis time is typically only a small part of the DFT analysis. The major cause is the internal state of DFT gates. For instance, for a PAND-gate one needs to keep track in which order children fail, while for a SPARE-gate the status of the spare components (are they in use or not, or even failed themselves?) and the order of the spare components (in case a SPARE has more than single spare component) needs to be administered. This altogether may yield huge state spaces. The technique advocated in this paper aims at boosting this process by simplifying DFTs prior to their state-space generation.

State of the art. An effective technique to keep the state space generation process manageable is to generate the CTMC in a *compositional* manner. The idea to apply this principle to DFTs goes back to Boudali et al. [BCS10] and basically works as follows. For each leaf and each gate of the DFT a CTMC is generated. As those behaviours affect only a single gate or leaf, these state spaces are relatively small. The individual CTMCs are then composed in parallel. In order to do so, the CTMCs are equipped with actions that can be used to synchronise them with CTMCs of other components. (This yields so-called interactive Markov chains [Her02], a mixture of labelled transition systems and CTMCs.) The compositional state space generation is done by considering pairs of CTMCs, reducing their state space by means of bisimulation—a.k.a. lumping—and composing the result with the next CTMC. This process is continued until a single CTMC is obtained. In this way, the peak memory consumption is not the state space of the entire CTMC, but the maximal size of the pairs of lumped Markov chains. As shown in Boudali et al. [BCS10], this mostly yields a compact state space. This approach is supported by the tool DFTCalc [ABvdB+13]. The DFT measures of interest such as system reliability and availability can then be established by standard means such as probabilistic model checking techniques for CTMCs [BHHK03].

DFT rewriting. The key idea of the approach in this paper is to *rewrite* a DFT prior to the state space generation. The rewriting is aimed to simplify a DFT by turning it into an equivalent DFT with a smaller state space. For static FTs, such rewriting can simply be done by adopting rules from Boolean algebra, e.g., $(x \wedge y) \vee (z \wedge y)$ is equivalent to $(x \vee z) \wedge y$. BDDs in fact yield compact canonical representations in many cases, i.e., the reduction is effectively done using the BDD representation.

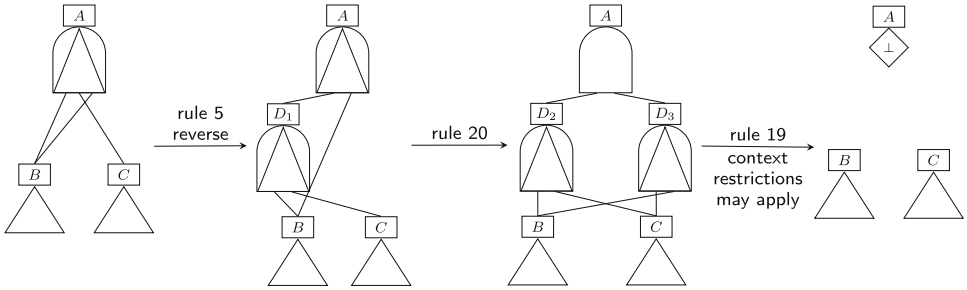


Fig. 1. An example of a simple rewriting of a DFT with conflicting ordering constraints

Rewriting DFTs is however more involved. This is due to the fact that the ordering of failures needs to be respected and the internal state of gates is relevant. To tackle these difficulties, we resort to *graph transformation*. The core idea is to consider DFTs as (typed) directed graphs and manipulate them by graph transformation [EEPT06], a powerful technique to rewrite graphs via pattern matching. A small example illustrates this idea. Consider the leftmost DFT in Fig. 1, where the top level event is a PAND-gate, and the children B and C represent arbitrary DFTs. This DFT is fail-safe, as the DFT requires B to fail before C, and C to fail before B. Our rewrite rules allow for rewriting this DFT into a DFT which is represented as $\bot$, see the rightmost DFT in Fig. 1. This is done in three rewrite steps. In the first step, the original PAND with three children is replaced by two binary PAND-gates. Though this introduces a new gate, it allows for a further simplification in the last step. The second rewrite step introduces an AND-gate as top event with two PAND-gates as child. This represents that the original DFT is indeed a conjunction of two conflicting orderings. The final rewrite step exploits this and results in the fail-safe DFT. As this rule is only correct provided the sub-DFTs B and C have no common elements, the latter rewrite step is subject to a context restriction.

We present a catalogue of 29 (families of) rules that rewrite a given DFT into a smaller, equivalent DFT. Here, equivalence means that the source and target DFT have the same system reliability and availability. Thus, the probability of the top level event failing within a given deadline as well as the mean time to failure for the top level event are preserved by our rewriting rules. In contrast to reduction rules for static FTs, various rewrite rules for DFTs are context sensitive and benefit from the powerful framework of graph transformations. (Even relatively simple rules like the elimination of children of an OR-gate that cannot fail, require context conditions). As the rewrite rules can often be applied in two directions, some guidance in the selection of rules is required. For instance, in the above example, it is of no use to apply the reverse of the first rewrite rule on the second (from the left) DFT. Note that our rewrite system is not strongly normalising, i.e., it is not the case that every sequence of rewrites terminates with an irreducible DFT. We use a simple heuristic which determines the order to apply the rewrite rules. Our empirical evaluation shows that this simple heuristic is already very effective. Figure 2 provides an overview of how DFT rewriting exploits standard graph transformation.

Empirical evaluation. We have implemented our rewrite technique by exploiting several existing tools. All our DFT rewrite rules are realised using the graph transformation tool GROOVE [GdMR+12]. Using the aforementioned heuristic, GROOVE takes an input DFT and rewrites it in a number of steps. The resulting DFT is analysed using the DFT analysis tool DFTCalc [ABvdB+13] which exploits the compositional state space generation and minimisation mentioned earlier. The analysis of the resulting CTMC is done with the model checker MRMC [KZH+11]. This together yields a fully automated tool chain for graph-based DFT reduction and analysis of common properties such as system reliability and availability. We have analysed several variations of seven benchmarks, comprised of over 170 DFTs in total, originating from standard examples from the literature as well as industrial case studies from aerospace and railway engineering. *Rewriting enabled to cope with 49 DFTs that could not be handled before*. For the other fault trees rewriting pays off, being much faster and more memory efficient, *up to two orders of magnitude*. This applies to both the peak memory footprint and the size of the resulting Markov chain (see Fig. 31b, c, page 48). This comes at no run-time penalty: graph rewriting is very fast and the CTMC generation is significantly accelerated due to the DFT reduction.

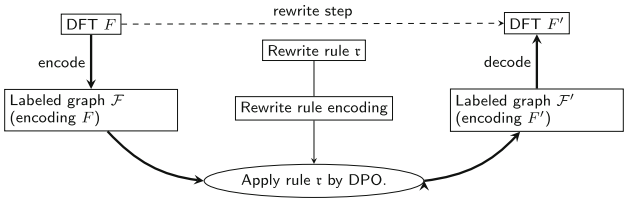


Fig. 2. Formalise DFT rewriting via standard graph transformation [EEPT06]

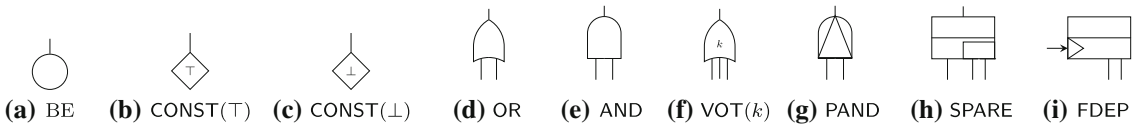


Fig. 3. Dynamic fault tree elements

Summary of the main contributions. The major contributions of this paper are:

- A rewriting framework for DFTs that allows for simplifying DFTs prior to their state-space generation.
- 29 (families of) rewrite rules for DFT rewriting together with their correctness.
- A prototypical implementation using graph transformation techniques and tools.
- An extensive experimental evaluation showing the potential benefits and state-space reductions.

This paper extends our conference paper [JGK+15] by a comprehensive presentation of all 29 (families of) rewriting rules, their correctness, and details about realising DFT rewriting as a standard graph transformation system. Full correctness proofs can be found in [Jun15].

Related work. Reduction of fault trees is a well-investigated topic. An important technique is to identify independent static sub-trees [PD96, LXL+10, HGH11, Yev11, RGD10]. These static sub-trees are analysed using efficient techniques (such as BDDs), whereas the dynamic parts require more complex methods. While such modular approaches yield significant speed ups, they largely depend on the DFT shape. Shared sub-trees, or a dynamic top-node, inhibits the application of these techniques. Merle et al. [MR07, MRLB10] map DFTs onto Boolean algebra extended with temporal operators. The resulting expressions can then be minimised via syntactic manipulation. This approach imposes several restrictions on the DFTs. Parametric fault trees [Rai05, BFGP03] exploit the symmetry in replicated sub-trees while translating DFTs (using graph transformation) to generalised stochastic Petri nets.

Organisation of the paper. Section 2 introduces DFTs, briefly describes their underlying CTMC, and provides several formal definitions relevant for the DFT rewriting. Section 3 introduces the main ingredients of graph rewriting. Section 4 explains the rewriting of DFTs such as context restrictions, DFTs as labelled graphs, what is a DFT rewrite rule, and well-formedness. Section 5 presents the 29 rewrite rules in detail and discusses their correctness. Section 6 describes how the DFT rewriting using graph transformation is realised in the graph transformation tool GROOVE. Section 7 presents the benchmark DFTs, provides the experimental results indicating the memory and time gains of DFT rewriting. Finally, Sect. 8 concludes the paper. Appendix A contains a table of often used notations. In Appendix B we briefly recap some formal definitions of graph rewriting.

2. Dynamic fault trees

Fault trees (FTs, [SVD+02]) model how individual component failures propagate through a system and lead to system failures. Since subtrees can be shared, FTs are directed acyclic graphs (DAGs) rather than trees. The leaves of the tree (or rather, the sinks of the DAG) model basic component failure modes, and are called *basic events (BEs)*. Fault tree *gates* model how component failures propagate and lead to system failures.

2.1. Static fault trees

To model a component's failure behaviour, each BE is usually equipped with a *failure rate* λ, i.e., the parameter of an exponential probability distribution. The probability that the BE fails before time T is then given by $1 - e^{-\lambda T}$. Other probability distributions, like Weibull, are often supported as well. We call those BEs whose failure behaviour is governed probabilistically *event elements (EEs)*. Some BEs have already failed or never fail. Such BEs are called *constants*, and denoted by $\mathsf{CONST}(\top)$ and $\mathsf{CONST}(\bot)$ respectively. Fig. 3a–c depicts these BE types.

The non-sink nodes of the FT, i.e., nodes with one or more children (a.k.a. inputs of the node), are equipped with *gates*. Static fault trees (SFTs) feature three types of gates, depicted in Fig. 3d–f:

- The AND-gate fails if all its children fail;
- The OR-gate fails if at least one of its children fails;
- The $\mathsf{VOT}(m)$-gate fails if at least m of its children fail.

Clearly, an OR-gate is equivalent to a $\mathsf{VOT}(1)$-gate, and an AND-gate with m children to a $\mathsf{VOT}(m)$-gate. Some SFT variants consider additional gates, like the XOR (exclusive OR) and the NOT-gate [SVD+02], which yield non-coherent behaviour and is only required in very special scenarios [CCR08].

A fault tree fails if its root, called the *top level event (TLE)*, fails. The TLE represents the failure condition of interest, such as the stranding of a train, or the unplanned unavailability of a data center, and is denoted by an underlined identifier. The failure of an SFT is determined by which BEs fail, and not by their order; hence their name *static*.

2.2. Dynamic fault trees

Static FTs appeal as a relatively simple, yet useful modelling tool. However, SFTs however cannot model several important failure patterns, such as: (a) *Order-dependent failures*, i.e. failures that only occur if BEs occur in a particular order. For example, a water leakage in a pump only leads to a short-circuit if the power supply has not failed before. (b) Simple support for *feedback loops*. Although systems with feedback loops in the error behaviour and cascaded errors can be modelled with SFTs, doing so requires verbose work-arounds which makes modelling error prone, cf. [SVD+02]. (c) *Spare management and spare parts.* A spare part is an interchangeable part that is used for the replacement of failed elements, e.g. spare car wheels. *Cold* spare parts can only fail while used. *Warm* spare parts can always fail but when being spare they fail with a reduced failure rate. *Hot* spare parts always have the same failure rate. SFTs can model hot spares via voting gates, but cold and warm spare cannot be represented.

To overcome these shortcoming *dynamic fault trees (DFTs)* were introduced, featuring three additional gates, see Fig. 3g–i:

- The (SPARE)-gate to model the management of spare parts;
- The priority-AND (PAND)-gate to model order dependent failures;
- The functional dependency (FDEP)-gate to model dependencies between failures and common cause failures.

These gates are called *dynamic*, since, unlike the static FTs, the order in which the BEs fail matters. Again, other dynamic gates exist, like the priority-OR, the sequence enforcer and the probabilistic dependency gate, see also [JGKS16]; we treat the most common ones here. Their behaviour is described below.

2.2.1. Basic events

An important property of spare management is that the failure behaviour of a spare depends whether or not it is in use. Components that are in use are called *active*; components not in use are *dormant* or *passive*. To accommodate this behaviour, DFTs do not only equip their BEs with a failure rate λ, as for SFTs, but also with a *dormancy factor* $\alpha \in [0, 1]$. The latter indicates how much the failure rate of a spare reduces when not in use: The probability for the BE to fail before time T is given by $1 - e^{-\lambda T}$ when active, and by $1 - e^{-\alpha \lambda T}$ when dormant. Thus, for a cold spare, we set $\alpha = 0$, so that it does not fail when dormant; a hot spare is given by $\alpha = 1$, so that it fails with the same rate no matter whether active or dormant; for a warm spare we set $\alpha \in (0, 1)$, so that it can fail when dormant, but with a lower rate $\alpha\,\lambda$.

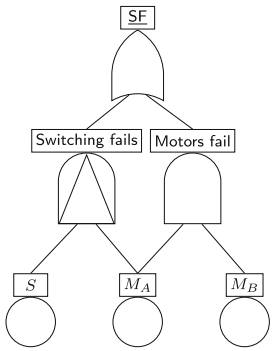


Fig. 4. Boom barrier example

As before, we label DFT leaves by the constant CONST($\top$) and CONST($\bot$) to denote respectively that have already failed or never fail. Finally, leaves that only fail due to an FDEP are not labeled at all.

2.2.2. Dynamic gates

The PAND**-gate.** A PAND-gate fails if its children fail from left to right; if the children fail in any other order, then the PAND-gate becomes *infallible*.

Example 2.1 Figure 4 depicts a very typical usage of the PAND-gate to model the switching between two redundant components. This DFT models a the drive train of boom barrier. The system has two redundant motors (M_A and M_B). A switch (S) connects either of the motors to the drive train; initially, M_A is connected to S. As soon as M_A fails, motor M_B can take over, but then the switch must first connect M_B to the barrier. This is only possible if the switch has not failed. Thus, the failure of the switch is only relevant if it occurs before M_A fails. The DFT reflects this behaviour as it only fails if either both motors fail, or if the switching fails, where the latter only happens if first the switch and then M_A fails.

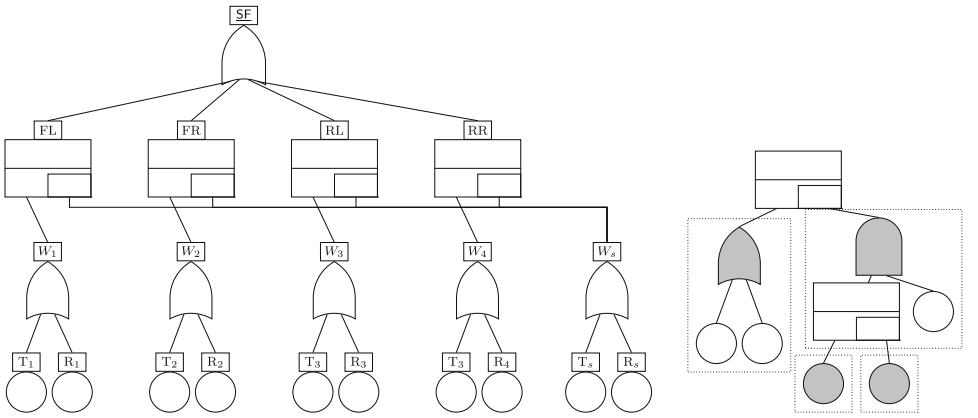
Being DAGs rather than trees, one BE or gate may serve as input to several gates. Hence, it may happen that several children of a PAND-gate fail simultaneously. Different interpretations of the PAND-gate exist as to whether or not the PAND should fail in that case [JGKS16]. We assume it does so, i.e. we take the non-strict interpretation here. That is, the PAND fails if all its children have failed, and each ith child fails before or at the same time as the i+1th child.

The SPARE**-gate.** A SPARE-gate models the management of spare components. The first child of a SPARE-gate is called its *primary*, and is in use from the start. The other children are called *spares* or *spare children* and replace the primary when it fails.

Example 2.2 A typical example is a car with a spare wheel in the trunk, see Fig. 5a. The car contains four spots for wheels (denoted FL, FR, RL and RR front-left, front-right, rear-left, and rear-right). Initially, wheels W_1 to W_4 are mounted, while the spare wheel W_s is in the trunk. Each wheel fails due to either a tire-fault (T_i) or a rim-fault (R_i). If (say) W_3 fails, then it is replaced by the spare wheel W_s. If any further wheel fails, then no spare wheel is left to replace W_3, so the system fails.

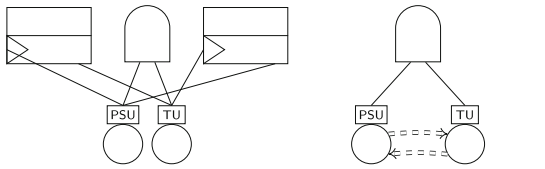
If an (either primary or spare) child of a SPARE-gate fails, then the SPARE-gate attempts to switch to the next (from left to right) available spare, i.e., a child that has not yet failed, and that is not in use by any other SPARE-gate. The process of obtaining the next available spare is called *claiming*. If a SPARE-gate has no available spare anymore, it fails.

In order to describe activation, we group elements in so-called *(spare-)modules*. Modules are subgraphs represented by a child of a SPARE-gate. In Fig. 5, we have in grey the representative of the sets of elements indicated by dotted boxes. In a module, all elements have the same activation status, and they are all activated at once. The precise extent of the modules is described in Sect. 2.4.4. We also construct a top module, represented by the TLE. Initially, the top module is active, furthermore, for each active SPARE-gate also the module represented by its used child is active. All BEs in an active module fail according to their active failure rate, all others fail according to their dormant failure rate.



(a) DFT for a spare wheel **(b)** Spare modules depicted

Fig. 5. SPARE-gate examples



(a) Gate notation **(b)** Arrow notation

Fig. 6. A feedback loop using two notations for FDEPs

The FDEP **gate.** The FDEP gate causes other elements to fail. The leftmost child is called a *trigger*, the other children are called *dependent events*. The behaviour of the FDEP is that, if the trigger fails, all dependent events automatically fail. In this way, FDEPs are used to represent common cause failures and feedback loops, such as a fire that causes all equipment at a particular location to fail.

Example 2.3 The DFT in Fig. 6a shows a feedback loop between a power supply unit (PSU) and a thermal unit (TU), modelling their mutually dependent failure: a failure of the PSU is leads to the failure of the TU, and vice versa.

By introducing FDEPs we now have two ways of passing failures between elements:

1. *Failure propagation* from an element to its parents (as in SFTs), or
2. *Failure forwarding* from an element to an EE (by FDEPs).

We assume that failure propagation takes presence over failure forwarding. That is, first we take into account the results of failure propagation, and only then consider if there is an opportunity for failure forwarding to let a dependent event fail: if so, we again first do the failure propagation. Put it differently, we only forward failures if no gate can fail anymore.

The presence of FDEPs yields two reasons for EEs to fail:

1. (*Internal*) *failure forwarding* an EE is triggered by an FDEP and has not failed before, or
2. *External failure causes* an EE fails due to a basic event and has not failed before.

Failure forwarding is ordered, but non-deterministic. That is, if one trigger is connected (via multiple FDEPs) to multiple dependent events, then the order in which the dependent events fail is not specified, i.e., any order can occur. If these dependent events are connected by a PAND-gate, then choosing a particular failure order influences whether or not the TLE fails, and therefore the quantitative measures, like the reliability. One can, however, study best-case versus worse-case scenarios, see Sect. 2.3.2.

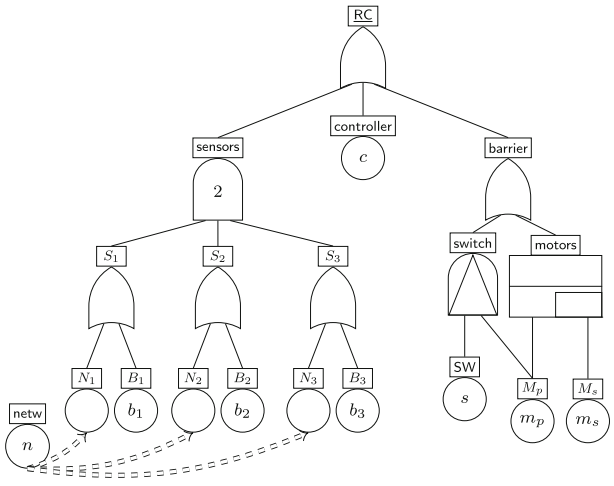


Fig. 7. Variant of the railway crossing DFT

By connecting the same trigger to multiple FDEP-gates, we can assume w.l.o.g. we that all FDEPs are binary, i.e. contain a single dependent event. To simplify the figures, we depict FDEPs by double-dotted arrow from the trigger to the dependent event, rather than by a (more explicit) gate; see Fig. 6a versus b.

Example 2.4 The DFT in Fig. 7 represents a simplified railway level crossing [GKS+14]. The system consists of three sensors, barriers as in Fig. 4, and a controller and fails if either of these subsystems fails. The sensor system fails if at least two of the three redundant sensors (S_1, S_2, S_3) fail. Sensor S_i fails either due to their battery (B_i) or due to a failed network cable (N_i). Furthermore, a network problem (netw) causes all sensor network connections N_i to fail, as modelled by the three FDEP-gates. Finally, the barrier fails if either the main and spare motor fail, modelled by the SPARE-gate Motors, or if the switch and then a motor fails, as modelled by the PAND-gate.

As basic events, we consider failures of the network (n), the batteries (b_1, b_2, b_3), the controller (c), the switch (s), or the motors (m_p, m_s). We attach these basic events to the corresponding EEs, indicated by placing the basic events inside the circles.

We do not consider basic events for the network failure of the sensors. The only way that the EEs N_1, N_2, and N_3 can fail is thus internally.

2.2.3. Well-formedness

In order to be meaningful, DFTs must adhere to some well-formedness conditions:

1. The DFT is acyclic;
2. VOT(k)-gates have at least k children;
3. The TLE is not an FDEP;
4. FDEPs have no parents;
5. All dependent events of an FDEP are event elements;
6. Spare modules, i.e., sub-trees under a SPARE-gate, are independent;
7. Primary spare module representatives, i.e., first children of a SPARE-gate, do not contain any CONST($\top$) elements; and
8. Primary spare modules are not shared over multiple SPARE-gates.

The conditions (1–5) are standard and self-explanatory. The independence of two spare modules (6) prevents ambiguity in the meaning of claiming spares. Constraint (7) enables us to simplify the definition of the initially claimed elements. Constraint (8) again simplifies matters: primary modules are initially claimed by a unique SPARE-gate, and therefore they cannot be claimed later on and sharing them is superfluous.

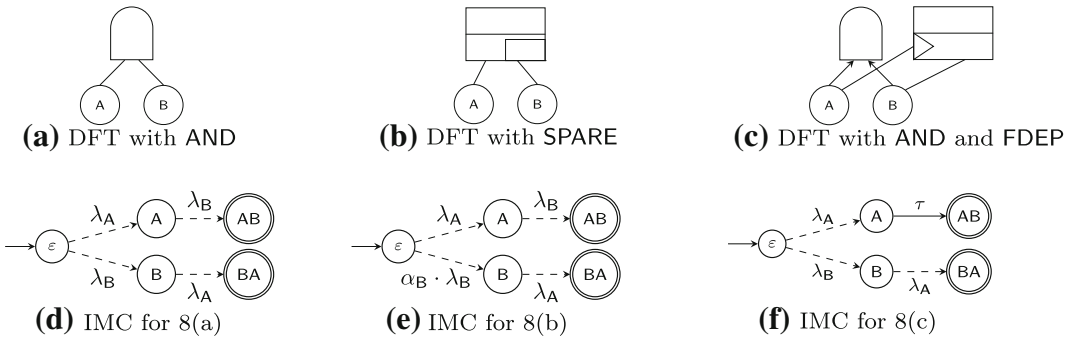


Fig. 8. Three simple DFTs and their state-transition semantics

2.3. DFT analysis

2.3.1. The state space of a DFT

The semantics of a DFT F is—like in [CSD00]—expressed using a state-transition system $\mathcal{C}_F$, where transitions correspond to the failure of an EE and states to sequences of distinct EEs. As our interest is in quantitative analysis of DFTs, transitions are labeled with the failure rate of an EE. The triggering of events using FDEPs (internal failures) is instantaneous and is instead modelled by a transition that is labelled by an action (denoted τ). The resulting state-transition system $\mathcal{C}_F$ thus has two types of transitions: rate-labeled and τ-labelled transitions. This is known as an interactive Markov chain (IMC, for short) [Her02]. The detailed construction of $\mathcal{C}_F$ is given in Sect. 2.4. Here, we illustrate the DFT semantics by three small examples, see Fig. 8. For ease of reference, we labelled each state with the sequence of failed EEs. The DFT in Fig. 8a fails if EE A fails before B, or B and A fail in the reverse order, see Fig. 8d. The DFT in Fig. 8b fails if EEs A and B fail, as above. However, if A has not yet failed, then the failure rate of B is reduced by the dormancy factor α_B. Thus, B is initially dormant, and fails with rate $\lambda_B \cdot \alpha_B$ (see Fig. 8e). The DFT in Fig. 8c also fails if EEs A and B fail. However, the failure of A causes B to fail immediately afterwards, as realised by the τ-transition in Fig. 8f.

2.3.2. DFT measures

We define the quantitative measures on a DFT F in terms of the IMC $\mathcal{C}_F$. Let Fail be the set of states in IMC $\mathcal{C}_F$ in which the *top* event in F has failed. Recall that, if one FDEPs trigger triggers several dependent events, their order is non-deterministically resolved. This is formalised using a scheduler (or policy) $\mathcal{S}$ on $\mathcal{C}_F$, resulting in a continuous-time Markov chain $\mathcal{C}_F[\mathcal{S}]$.

Definition 2.1 (*Quantitative* DFT *measures*) Let F be a DFT, $\mathcal{S}$ a scheduler on F and $t \in \mathbb{R}$.

- $\mathsf{RELY}^{\,t}_{F[\mathcal{S}]} = \Pr_{\mathcal{C}_F[\mathcal{S}]}(\Diamond^{\leq t}\,\mathsf{Fail})$ is the *reliability of F under $\mathcal{S}$ given mission time t*.
- $\mathsf{MTTF}_{F[\mathcal{S}]} = \mathsf{ET}_{\mathcal{C}_F[\mathcal{S}]}(\Diamond\,\mathsf{Fail})$ is the *mean time to failure in F under $\mathcal{S}$*.

We define the *reliability of F given mission time t* as tuple $\mathsf{RELY}^{t}_{F} = (\min_{\mathcal{S}} \mathsf{RELY}^{\,t}_{F[\mathcal{S}]}, \max_{\mathcal{S}} \mathsf{RELY}^{\,t}_{F[\mathcal{S}]})$ and the *mean time to failure in F* as tuple $\mathsf{MTTF}_F = (\min_{\mathcal{S}} \mathsf{MTTF}_{F[\mathcal{S}]}, \max_{\mathcal{S}} \mathsf{MTTF}_{F[\mathcal{S}]})$

Here, $\Pr(\Diamond^{\leq t}\,\mathsf{Fail})$ stands for the probability of all paths that reach a state in Fail ultimately at time t, whereas $\mathsf{ET}(\Diamond\,\mathsf{Fail})$ stands for the expected time until reaching a state in which the top element of the DFT has failed. These notions can be defined on the IMC in a standard way and go beyond the scope of this paper. For more information, we refer to [GHH+14].

We focus on quantitative measures as their analysis is typically most time consuming. The rewriting of many other quantitative measures can also be defined on the IMC. Most rewriting rules are shown to preserve the IMC up to isomorphism—which is also the correctness argument used (cf. Sect. 5.4) for these rules, therefore, using only the adequate subset of rules preserves most quantitative measures. Regarding qualitative measures, one can similarly use a subset of rules which preserve the property at hand. Example measures for the DFT in Fig. 7 are: "what is the reliability of the level crossing for a time frame of 10 years after deployment?" (reliability), or "what is the mean time until a first failure of the level crossing occurs?" (MTTF).

2.4. Formal definitions

2.4.1. DFT syntax

To formally describe DFTs, we assume a fixed set BasicEvents containing all basic events, which are assigned a failure rate and dormancy factor via the function Λ

$$\Lambda : \mathsf{BasicEvents} \to \mathbb{R}_{>0} \times [0, 1]$$

DFT nodes are referred to as *elements*. They are partitioned into *leaves* and *gates*.

$$\mathsf{Leaves} = \{\mathsf{EE}, \mathsf{CONST}(\top), \mathsf{CONST}(\bot)\}$$
$$\mathsf{Gates} = \{\mathsf{AND}, \mathsf{OR}, \mathsf{PAND}, \mathsf{FDEP}, \mathsf{SPARE}\} \cup \{\mathsf{VOT}(k) \mid k \in \mathbb{N}\}$$

Recall that event elements (EEs) fail according to a probability distribution, CONST($\bot$) leaves never fail and CONST($\top$) leaves have failed already.

Definition 2.2 (*Dynamic fault tree*) Formally, a DFT F is a tuple $(V, \sigma, Tp, \Theta, top)$, where

- V is a finite set of *nodes*.
- $\sigma\colon V \to V^*$ maps each node to its (ordered) *successors*, also called *children* or *inputs*.
- $Tp\colon V \to \mathsf{Gates} \cup \mathsf{Leaves}$ is a *typing*, mapping leaf nodes ($v \in V$ with $\sigma(v) = \epsilon$) to Leaves and other nodes to Gates. For $K \in \mathsf{Gates} \cup \mathsf{Leaves}$, let $F_K = \{v \in V \mid Tp(v) = K\}$ be the nodes of type K.
- $\Theta\colon F_{\mathsf{EE}} \nrightarrow \mathsf{BasicEvents}$ is the (partial) injective *attachment* function.
- $top \in V$ is the unique *top* element.

We often write "v is a K" for $v \in F_K$. The attachment function Θ is partial, i.e., some event elements have no associated BasicEvents, allowing a more modular simplification of the FDEP in Sect. 5.3. For node v of a DFT with children $\sigma(v) = v_1 \cdots v_n$, let $\sigma(v)_i = v_i$ denote the ith input of v, $\mathsf{child}(v) = \{v_1, \ldots, v_n\}$ the set (rather than the sequence) of all inputs, and $\mathsf{parent}(v) = \{v' \mid v \in \mathsf{child}(v')\}$ the set of v's predecessors.

Example 2.5 The DFT in Fig. 7 is formally given as $(V, \sigma, Tp, \Theta, top)$ with

- $V = \{\mathsf{RC}, \mathsf{sensors}, \mathsf{controller}, \mathsf{barrier}, \mathsf{switch}, \mathsf{motors}, \mathsf{SW}, M_p, M_s, S_1, S_2, S_3, \ldots, X_1, X_2, X_3\}$ where X_i denote the FDEPs.
- σ given by $\mathsf{sensors} \mapsto S_1, S_2, S_3$; $S_1 \mapsto N_1, B_1$; $N_1 \mapsto \varepsilon$; $X_1 \mapsto \mathsf{netw}, N_1$; $S_2 \mapsto N_2, B_2$; $N_2 \mapsto \varepsilon$; ...; where ε denotes the absence of successors.
- Tp given by $\mathsf{sensors} \mapsto \mathsf{VOT}(2)$; $S_i \mapsto \mathsf{OR}$, $N_i \mapsto \mathsf{EE}$, $B_i \mapsto \mathsf{EE}$ for all $i \in \{1, \ldots 3\}$; $X_1 \mapsto \mathsf{FDEP}$; $X_2 \mapsto \mathsf{FDEP}$; ....
- Θ given by $\mathsf{netw} \mapsto n$; $B_1 \mapsto b_1$; $B_2 \mapsto b_2$; $B_3 \mapsto b_3$; $\mathsf{SW} \mapsto s$; ... and undefined on N_1, N_2, N_3.
- $top = \mathsf{RC}$.

2.4.2. The graph underlying a DFT

Each DFT F induces an underlying graph G_F whose edges point from a node to its children. That is, (v, v') is an edge if v' is a child of v. Thus, paths in G_F "walk down the tree," going from higher to lower nodes in F. The undirected graph $G_F^{\leftrightarrows}$ contains edges from parent to child and from child to parent.

Definition 2.3 (*Underlying graph of a* DFT) The *simple graph* of DFT $F = (V, \sigma, Tp, \Theta, top)$ is $G_F = (V, E)$ where $E = \{(v, v') \in V \times V \mid v' \in \mathsf{child}(v)\}$. Let $G_F^{\leftrightarrows} = (V, E \cup E^{-1})$ be the *undirected graph* of DFT F.

A sequence $v_0 e_1 v_1 \ldots e_n v_n \in V \times (E \times V)^n$ is a *path from* v_0 *to* v_n through a graph $G = (V, E)$ if $e_{i+1} = (v_i, v_{i+1})$ for all $0 \leq i < n$. The set $\mathrm{Path}_F(x, y)$ denotes all paths from x to y through $G_F^{\leftrightarrows}$.

We define the set of descendants $\mathsf{dec}(v)$ of a node $v \in V$ as all nodes v' that can be reached from v via a path in the underlying graph G_F. The set of ancestors $\mathsf{anc}(v)$ contains all nodes v' from wich v can be reached in G_F.

$$\mathsf{dec}(v) = \{v' \in V \mid \text{there exists a path from } v \text{ to } v' \neq v \text{ in } G_F\} \quad \text{and}$$
$$\mathsf{anc}(v) = \{v' \in V \mid \text{there exists a path from } v' \neq v \text{ to } v \text{ in } G_F\}.$$

Example 2.6 Consider the DFT in Fig. 7. We have

- $\mathsf{child}(\text{sensors}) = \{S_3, S_1, S_2\}$ (that is, an unordered set),
- $\mathsf{parent}(M_p) = \{\text{switch}, \text{motors}\}$,
- $\mathsf{dec}(\text{sensors}) = \bigcup_{1 \leq i \leq 3} \{S_i, B_i, N_i\}$,
- $\mathsf{anc}(M_p) = \{\text{switch}, \text{motors}, \text{barrier}, \text{RC}\}$.

2.4.3. State of a DFT

We use the following notations. Let $\mathcal{P}(A)$ denote the powerset of a set A and $\mathcal{P}_k(A)$ denote all subsets of A of cardinality k. We use K^* to denote the set of (finite) words over alphabet K where ε denotes the empty word and $K^{\triangleright}$ are the words over K without repetition.

As described in Sect. 2.3.1, the state of DFT F depends on the events that have failed and their ordering. We use $F_{\mathsf{EE}}^{\triangleright} \subseteq F_{\mathsf{EE}}^{*}$ to denote the set of ordered sequences of event element failures and π for elements in $F_{\mathsf{EE}}^{\triangleright}$. Then the state of the DFT F after the subsequent occurrence of the events in π is characterised by two predicates, $\text{Failed}(\pi)$ and $\text{ClaimedBy}(\pi)$. The first predicate describes which elements are considered failed after π occurred, the second maps spare module representatives to SPARE-gates that claimed them. The DFT semantics guarantees that each spare module is claimed by at most one SPARE-gate.

To define these predicates, we use a set of relations. For DFT $F = (V, \sigma, Tp, \Theta, top)$ let $\models_F \subseteq V \times F_{\mathsf{EE}}^{\triangleright}$ be the *model-relation*, and $\dagger_F^s \subseteq V \times F_{\mathsf{EE}}^{\triangleright}$ be the *claiming-relation* for SPARE-gate s in F. The model-relation describes whether a node $v \in V$ fails for a given event trace $\pi \in F_{\mathsf{EE}}^{\triangleright}$, and the claiming-relation describes whether a node $v \in V$ is claimed by a spare for a given event trace $\pi \in F_{\mathsf{EE}}^{\triangleright}$.

Definition 2.4 (*Failed events after a trace*) Let $F = (V, \sigma, Tp, \Theta, top)$ be a DFT and $\pi \in F_{\mathsf{EE}}^{\triangleright}$. The *set of failed elements after* π is defined as

$$\text{Failed}\colon F_{\mathsf{EE}}^{\triangleright} \to \mathcal{P}(V),$$
$$\text{Failed}(\pi) = \{v \in V \mid v \models_F \pi\},$$

and the mapping of *representatives claimed by after* π as

$$\text{ClaimedBy}\colon F_{\mathsf{EE}}^{\triangleright} \to (V \to \mathcal{P}(F_{\mathsf{SPARE}})),$$
$$\text{ClaimedBy}(\pi)(v) = \{s \in F_{\mathsf{SPARE}} \mid v \dagger_F^s \pi\}.$$

Example 2.7 Consider the DFT in Fig. 7. Let $\pi = \langle \mathsf{SW}, M_p \rangle$ and $\pi' = \langle M_p, \mathsf{SW} \rangle$. Then $\text{Failed}(\pi) = \{\mathsf{SW}, \mathsf{switch}, \mathsf{barrier}, \mathsf{RC}\}$, whereas $\text{Failed}(\pi') = \emptyset$. The difference in the traces π and π' is the order of failed EEs. For $\mathsf{switch} \in F_{\mathsf{PAND}}$, the model relation asserts that the node fails if all children fail in order. Thus, $\mathsf{switch} \models_F \pi$ but $\mathsf{switch} \not\models_F \pi'$. The claiming-relation is only interesting for the spares, where $\text{ClaimedBy}(\cdot)(M_p) = \{\mathsf{Motors}\}$ for π and π'.

2.4.4. FDEPs, modules and activation

Functional dependencies. Once a trigger event fails, the FDEP-gate causes the dependent event element to fail. The direct relation of functional dependencies by dependent events is defined as:

$$\text{DepEvents}(v) = \{\sigma(w)_i, i > 1 \mid w \in \mathsf{FDEP}, \sigma(w)_1 = v\}.$$

The set of *dependent* or *triggered events* of trace π is given by:

$$\text{DepEvents}(\pi) = \{w \notin \pi \mid v \in \text{Failed}(\pi) \text{ and } w \in \text{DepEvents}(v)\}.$$

States with $\text{DepEvents}(\pi) \neq \emptyset$ only have outgoing τ-transitions.

Example 2.8 Consider the DFT in Fig. 7. We have DepEvents(netw) $= \{N_1, N_2, N_3\}$ and DepEvents(N_1) $= \emptyset$. Furthermore, for $\pi = \langle N_1 \rangle$, DepEvents($\pi$) $= \emptyset$ and for $\pi = \langle N_1, \text{netw} \rangle$ we have DepEvents(π) $= \{N_2, N_3\}$.

Spare modules. Spare modules are sets of elements that are claimed and activated together. Each module is represented by a representative r, which is a child of a SPARE-gate. As functional dependencies are not supposed to propagate activation, elements which are only connected via functional dependencies are separate modules. Furthermore, children of a spare are usually another spare module, thus the paths connecting SPARE-gates to their children should not define a connection of elements. Modules consist of sets of elements which are connected via *module paths*.

Definition 2.5 (*Module path*) A path $p = v_0 e_1 v_1 \dots e_n v_n$ in $G_F^{\leftrightarrows}$ of DFT $F = (V, \sigma, Tp, \Theta, top)$ is a *module path* from v_0 to v_n if the following conditions hold

- $v_i \notin V_{\mathsf{FDEP}}$, for all $0 \leq i \leq n$,
- $v_i \in V_{\mathsf{SPARE}} \Rightarrow v_{i+1} \in \mathsf{parent}(v_i)$, for all $0 \leq i < n$,
- $v_i \in V_{\mathsf{SPARE}} \Rightarrow v_{i-1} \in \mathsf{parent}(v_i)$ for all $1 \leq i \leq n$.

The set of module paths from $v \in V$ to $v' \in V$ is denoted by $\text{ModPath}_F(v, v')$.

Modules are represented by the children of SPAREs or by the *top* element—every element connected via a module path to a representative is in the same module: Let the set of *module representatives* be defined as $(\bigcup_{s \in F_{\mathsf{SPARE}}} \mathsf{child}(s)) \cup \{top\}$. Given a module representative r, the *module represented by* r is defined as $\mathsf{Mod}_r = \{v \in V \mid \text{ModPath}_F(r, v) \neq \emptyset\}$. If $r = top$, the corresponding module is called the *top module*, otherwise it is called a *spare 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 a *primary module*.

Example 2.9 Consider the DFT in Fig. 5a. Each set $\{W_i, T_i, R_i\}$, for $i = 1, 2, 3, 4$, s, is a module with representative W_i. Sets W_1, W_2, W_3 and W_4 are primary modules. The top module {SF, FL, FR, RL, RR} is represented by SF.

Remark **1** Some models do not have strictly separated spare modules; which is unproblematic due to the following observation (formally discussed in [BCS10]): Any element connected via a module path to the root element is initially activated. Likewise, elements connected to the primary children of non-nested SPARE-gates are initially activated. Thus, the notion of modules is not necessary for primary activated modules. As an example, consider the DFT depicted in Fig. 7, where m_p is both connected to a top and (trivially) to the primary child of the SPARE-gate. For simplicity, we do not formally allow this construction here as it overcomplicates the definitions. Notice that this does not have any impact, as (i) the construction are simply syntactic sugar and (ii) such occurrences can be easily detected.

Activation. The module represented by *top* is always active. Basic events are activated whenever they are attached to event elements which are part of an active module. Modules are activated when their representative is claimed by an active spare, or the spare which claimed their representative becomes active. To describe which event elements are active, let Active be defined as the least fix-point such that:

$$\text{Active}\colon F_{\mathsf{EE}}^{\triangleright} \to \mathcal{P}(V),$$

$$\text{Active}(\pi) = \mathsf{Mod}_{top} \cup \bigcup \{\mathsf{Mod}_r \mid \text{ClaimedBy}(\pi)(r) \subseteq \text{Active}(\pi)\}.$$

A basic event is *activated* whenever its corresponding EE is active:

$$\text{Activated}(\pi) = \{\omega \in \mathsf{BasicEvents} \mid \exists\, v \in F_{\mathsf{EE}}\ \Theta(v) = \omega \wedge v \in \text{Active}(\pi)\}.$$

Example 2.10 Consider the DFT in Fig. 7. We have Active(ε) $= V \setminus \{\text{netw}, M_s\}$. For $\pi = \langle M_p \rangle$ we get Active(π) $= V \setminus \{\text{netw}\}$. We get Activated($\varepsilon$) $= \{b_1, b_2, b_3, s, c, m_p\}$ and Activated(π) $=$ Activated(ε) $\cup \{m_s\}$.

3. Graph rewriting

The previous section has provided preliminaries and formal definitions of DFTs and related concepts. This section focuses on the second main ingredient of this paper: graph rewriting. As we exploit well-known results from graph

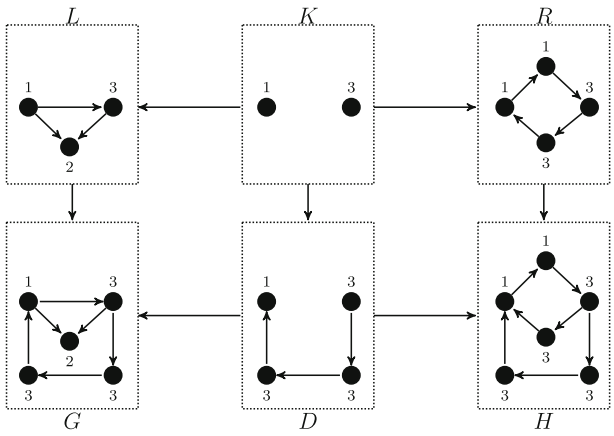


Fig. 9. Example for DPO graph rewriting [EEPT06]

rewriting, we give a brief introduction into graph rewriting and fix some notations. More detailed accounts on graph rewriting can be found in [EEPT06, Hec06]. We ensured that the essential results of this paper can be followed without a deeper understanding of graph rewriting theory.

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, as introduced by Ehrig et al. in [EPS73]. Essentially, a *rule* consists of two graphs, L and R, with a common *interface* K, which can roughly be thought of as a subgraph of both L and R—with a caveat discussed after the example below. *Rewriting* of a host graph G using the rule is done by finding a subgraph in G which matches L; this boils down to finding a mapping from L to that subgraph. G is then rewritten by (roughly) removing the image of $L \backslash K$ and adding an image of $R \backslash K$. Before giving a formal definition, we illustrate rewriting by an example from [EEPT06].

Example 3.1 We describe the application of a given rewrite rule on a given host graph. All graphs have vertex labels taken from $\{1, 2, 3\}$ and edge labels $\{x\}$. Figure 9 depicts the rule (graphs L, K, R) which is applied to the host-graph (G). We give an intermediate step (D) and the final result (H). We omit any edge labels in the figure. The arrows between the graphs depict the graph morphisms and are explained below.

Graph L describes the subgraph which has to be matched in a host graph. The graph K describes a subgraph of L referred to as the *interface*. After successfully matching L in G, elements which are matched by L but not by K are removed, yielding graph D. D has to be a valid graph, so dangling edges (i.e., edges without a source or target vertex) are not allowed. To prevent dangling edges in D, additional conditions are imposed on the match of L in G. In this particular example, we trivially match L in G, after which we remove all matched edges and the vertex labelled 2. This yields D.

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

In the example, the mapping from K to R is injective, and thus K is essentially a subgraph of R. However, in general, the relation is given by a possibly non-injective morphism. A non-injective morphism intuitively means that some nodes in D have to be merged to obtain H. Thus, if the morphism is non-injective, the second step, in addition to gluing R to D, also merges any nodes of D that are images of K-nodes with a common image in R. In other words, if nodes of K are mapped non-injectively into R, this induces a node merge in the rewrite step.

We apply graph rewriting on *labelled graphs*, digraphs whose vertices and edges are labelled. Let $\Sigma = \Sigma_v \uplus \Sigma_e$ be a (globally given) *label universe* consisting of disjoint sets of *vertex labels* Σ_v and *edge labels* Σ_e.

Definition 3.1 (*Labelled graph, graph morphism*) A *labelled graph* $G = (V, E, \mathsf{src}, \mathsf{tar}, \mathsf{lab})$ where

- V and E are disjoint finite sets of *vertices* V and *edges* E;
- $\mathsf{src}, \mathsf{tar} \colon E \to V$ are functions defining the *source* and *target* vertices, respectively;
- $\mathsf{lab} = \mathsf{lab}_v \uplus \mathsf{lab}_e$ is the *labelling*, with *vertex-labelling* $\mathsf{lab}_v \colon V \to \Sigma_v$ and *edge-labelling* $\mathsf{lab}_e \colon E \to \Sigma_e$.

Given two labelled graphs G and H, a *graph morphism* $m : G \twoheadrightarrow H$ is a function $m : (V_G \uplus E_G) \to (V_H \uplus E_H)$ that can be decomposed as $m = m_v \uplus m_e$ with label mapping $m_v : V_G \to V_H$ and edge mapping $m_e : E_G \to E_H$, which preserves edge sources, targets and labels: i.e., such that $m_v \circ \mathsf{src}_G = \mathsf{src}_H \circ m_e$, $m_v \circ \mathsf{tar}_G = \mathsf{tar}_H \circ m_e$ and $\mathsf{lab}_G = \mathsf{lab}_H \circ m$.

When this does not give rise to any confusion, we sometimes write "a morphism $G \twoheadrightarrow H$" without referring to the name of the morphism. Let $\mathrm{In}, \mathrm{Out} \colon V \to \mathcal{P}(E)$ denote the set of incoming and outgoing edges of a given node, respectively, and $\mathrm{InV}, \mathrm{OutV} \colon V \to \mathcal{P}(V)$ denote their source (target) vertices. These notations are used in Sect. 4.4.

$$\mathrm{In} \colon v \mapsto \{e \in E \mid \mathsf{tar}(e) = v\} \qquad \mathrm{Out} \colon v \mapsto \{e \in E \mid \mathsf{src}(e) = v\}$$
$$\mathrm{InV} \colon v \mapsto \{\mathsf{src}(e) \mid e \in \mathrm{In}(v)\} \qquad \mathrm{OutV} \colon v \mapsto \{\mathsf{tar}(e) \mid e \in \mathrm{Out}(v)\} \ .$$

A rewrite rule (the upper part in Fig. 9) is formalised as follows.

Definition 3.2 (*Graph rewrite rule*) A *graph rewrite rule* is a tuple $r = (L, K, R, K \twoheadrightarrow L, K \twoheadrightarrow R)$ with

- L, K, R are labelled graphs
- $K \twoheadrightarrow L, K \twoheadrightarrow R$ are morphisms, with $K \twoheadrightarrow L$ injective.

We call L the *left-hand side*, R the *right-hand side* and K the *interface* of rewrite rule r. We often abbreviate a rule by $(L \twoheadleftarrow K \twoheadrightarrow R)$. A rule is injective if $K \twoheadrightarrow R$ is injective.

For the purpose of this paper, it is important to realise that applying a graph rewrite rule r to a graph G consists of two steps:

1. Finding a match m of L, the rule's left-hand side graph, into G. Such a match is actually given by a graph morphism that satisfies the following two conditions (besides being a morphism):

No dangling edges For all $e \in E_G$ and all $v \in V_L$ *not in the image of* $K \twoheadrightarrow L$, if $m(v) = \mathsf{src}(e)$ or $m(v) = \mathsf{tar}(e)$ then $e = m(e')$ for some $e' \in E_L$.

No confusion for deleted elements For all $x \in V_L \uplus E_L$ *not in the image of* $K \twoheadrightarrow L$, there is no $y \in (V_L \uplus E_L) \backslash \{x\}$ such that $m(x) = m(y)$.

There may be more than one appropriate morphism between any two graphs, or none; so the step of finding a match is both non-deterministic and uncertain to succeed.

2. Building a new graph by deleting, creating and merging elements as dictated by the remainder of the rule. Given a match, this step is completely deterministic (up to the choice of the identity of fresh target nodes and edges, which choice however does not affect the structure of the target graph).

The formal definition of rule application (step 2 above) can be found in Appendix B.

4. A framework for rewriting dynamic fault trees

DFTs tend to be verbose. They are often based on the system architecture, therefore reflecting the sub-system structure [SVD+02]. Modern techniques automatically generate DFTs from architectural description languages [BCK+11], yielding rather verbose DFTs too. As state-of-the-art DFT analysis algorithms typically construct an underlying state space that is exponential in the size of the input DFT, it is a natural idea to shrink DFTs prior to their analysis. We do so by *rewriting* of DFTs.

This section describes in detail what it means to rewrite (well-formed) DFTs. We start off in Sect. 4.1 by showing that DFT rewriting is not context-free. Section 4.2 describes how to encode all information of a DFT in a labelled graph; this provides the basis for reducing DFT rewriting to graph rewriting. Section 4.3 then describes what a rewrite rule for DFTs is. Section 4.4 defines the semantics of a DFT rewrite rule. Finally, Sect. 4.5 discusses how to ensure that the result of DFT rewriting is well-formed. At the end of this section, we have all concepts in place to rewrite a (well-formed) DFT into another (well-formed) DFT. The formal relationship between these DFTs is discussed later in Sect. 5.

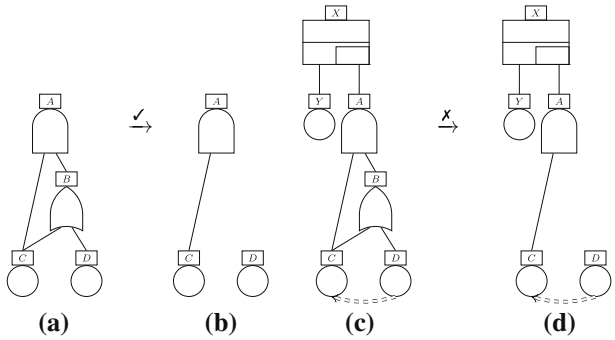


Fig. 10. The correctness of DFT rewriting depends on the context. Rewriting (**a**) into (**b**) is valid, while rewriting (**c**) into (**d**) is not, see Example 4.1

4.1. The need for context conditions

Simplifying *static* fault trees (SFTs) can be done by Boolean manipulations [SVD+02]. This remains true in the presence of FDEPs, but no longer holds for DFTs with dynamic gates such as SPAREs and PANDs, for which context-dependent rules are required.

Example 4.1 In the DFT in Fig. 10a, gate A fails if and only if C fails. It may seem a general rule that the OR-gate B does not contribute to the failure behaviour of A and can therefore be removed as a successor of A, as depicted in Fig. 10b. If, however, we apply the same reasoning in Fig. 10c, then we obtain an invalid result: by removing the path from A to D, we prevent the activation of D; in the DFT Fig. 10d, D is never activated. In particular, if D is in cold standby, then D never fails in Fig. 10d, and thus never triggers C, which leads to a higher reliability and MTTF than for the DFT in Fig. 10c. Since the our rewrite rules aim to preserve the reliability and MTTF, we disallow rewriting Fig. 10c to d.

4.2. Encoding DFTs as labelled graphs

In order to encode DFTs as graphs, we first consider the representation of DFT gates as graph vertices. DFTs are akin to labelled graphs, with one important difference: children of DFT gates are ordered. We use edge labels to encode this order in the graph encoding of a DFT. Furthermore, we cannot label vertices with the node type directly: untyped nodes in the left-hand side of the rule should match any (arbitrarily typed) node; yet matching nodes requires the same labels. Hence, we introduce an auxiliary vertex which carries the type information. Thus, the graph encoding of a DFT contains two to four vertices for each DFT node v: element vertex $(v, 1)$ corresponds to v; carry vertex $(v, 2)$ carries the type information for v; and, event element vertex $(v, 3)$ encodes the attachment Θ. The DFT gate *top* has an extra vertex $(v, 4)$. Graph vertices are labelled as follows: element vertices are labelled with a—meaningless—number 0; carry vertices are labelled by their type; EE vertices are labelled with the basic events as given by Θ.

The edges of the DFT encoding are defined as follows: as element vertices encode the DFT structure, there is an edge from $(v, 1)$ to $(v', 1)$ if v is a parent of v' in the DFT. There is also an edge between the carry vertex and (for EEs with an attached basic event) basic event vertices to the element vertices with edges from $(v, 1)$ to $(v, 2)$ and (if applicable) from $(v, 1)$ to $(v, 3)$. The edge labels are used to encode the order of the children. That is, the edge from $(v, 1)$ to $(v', 1)$ is labelled by an order-preserving mapping, defined below. All other edges, i.e. all edges to carry- and basic element vertices, are labelled by a—meaningless—number 0.

Example 4.2 Figure 11 depicts a DFT and its encoding as labelled graph. The encoding has three element vertices, with on their left side their carry vertices. Moreover, the topmost vertex has a carry vertex which identifies it as the top vertex, while the two basic elements have extra carry vertices for the attachment function. All carry vertices are connected via edges labelled with 0, while the other two edges have non-zero labels encoding the ordering.

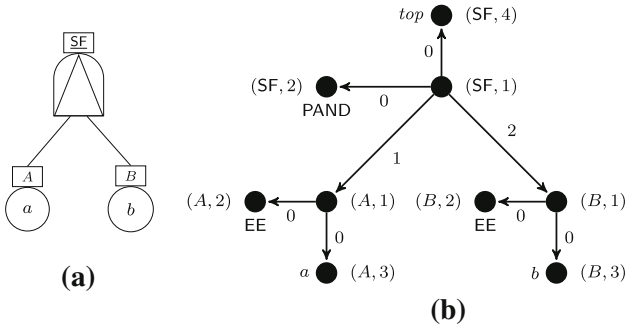


Fig. 11. DFT (**a**) and its labelled graph representation (**b**)

An edge-order preserving mapping constructs the edge labels in the order of the successor-sequence in a DFT. For encoding, we simply count the children from left to right; this is a trivial encoding. As rewriting may delete edges, we need a slightly more general definition for decoding.

Definition 4.1 (*Edge-order preserving*) Let $F = (V, \sigma, Tp, \Theta, top)$ be a DFT. A mapping $o\colon E(\sigma) \to \mathbb{N}$ is *edge order preserving* if for each $v \in V$ with $1 \leq i < j \leq |\ \mathsf{child}(v)\ |$ it holds that

$$o((v, \sigma(v)_i)) < o((v, \sigma(v)_j)).$$

If for all $v \in V$ and $1 \leq i \leq |\ \mathsf{child}(v)\ |$ we have $o(v, \sigma(v)_i) = i$, then we call o trivial.

Definition 4.2 (*Labelled graph of a* DFT) Let DFT $F = (V, \sigma, Tp, \Theta, top)$ and o be a trivial edge order preserving mapping. The *DFT graph representation for* F is the labelled graph $\mathcal{F} = (V_\mathcal{F}, E_\mathcal{F}, \mathsf{src}, \mathsf{tar}, \mathsf{lab})$ over $((\mathsf{BasicEvents} \cup \{0\} \cup \mathsf{Gates} \cup \mathsf{Leaves} \cup \{0, top\}), \mathbb{N})$ with

- $V_\mathcal{F} = (V \times \{1, 2\}) \cup (F_{\mathsf{EE}} \times \{3\}) \cup \{(top, 4)\}$,
- $E_\mathcal{F} = \{\langle (v, 1), (v', 1)\rangle \mid (v, v') \in E(\sigma)\}$
  $\cup\ \{\langle (v, 1), (v, 2)\rangle \mid v \in V\}$
  $\cup\ \{\langle (v, 1), (v, 3)\rangle \mid v \in F_{\mathsf{EE}}\}$
  $\cup\ \{\langle (top, 1), (top, 4)\rangle\}$
- $\mathsf{src}_\mathcal{F}(\langle w, w'\rangle) = w$
  $\mathsf{tar}_\mathcal{F}(\langle w, w'\rangle) = w'$
- The vertex labelling function lab_v is given by

$$\begin{aligned} (v, 1) &\mapsto 0 \\ (v, 2) &\mapsto Tp(v) \\ (v, 3) &\mapsto \begin{cases} \Theta(v) & \text{if } v \in \mathrm{Dom}(\Theta) \\ 0 & \text{else.} \end{cases} \\ (top, 4) &\mapsto top \end{aligned}$$

- The edge labelling function lab_e is given by

$$\begin{aligned} \langle (v, 1), (v', 1)\rangle &\mapsto o((v, v')) \\ \langle (v, 1), (v, 2)\rangle &\mapsto 0 \\ \langle (v, 1), (v, 3)\rangle &\mapsto 0 \\ \langle (top, 1), (top, 4)\rangle &\mapsto 0. \end{aligned}$$

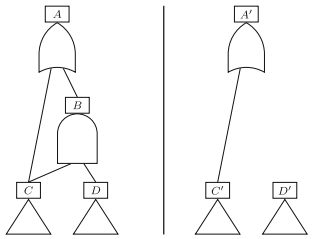


Fig. 12. The L and R subDFTs of the subsumption rule

4.3. Rewriting DFTs

The core of a DFT rewrite rule consists of the two (sub)DFTs, representing the left- and right-hand side of the rule. SubDFTs do neither contain a *top* node, nor contain an attachment function. These restrictions allow simplifications, while still allowing to rewrite DFTs into equivalent DFTs ranging over the same set of basic events. For example, Fig. 12 displays the left- and right-hand side for the rewrite rule from Example 4.1. Here, the EEs from the example are generalised to be of arbitrary type; this is indicated by a triangle. We refer to these rules as *structural rules* as they only change the structure of the DFT.

Definition 4.3 (*subDFT*) Let DFT $F = (V, \sigma, Tp, \Theta, top)$. A *subDFT of* F is a tuple $X = (V_X, \sigma_X, Tp_X^\sharp)$ with $V_X \subseteq V$, $\sigma_X = \sigma \mid V_X$ and $Tp_X^\sharp = Tp \mid Z$ where $Z \subseteq V_X$. $Y = (V_Y, \sigma_Y, Tp_Y^\sharp)$ is called a *subDFT of* F if it is a subDFT of some DFT F; Y is well-formed if it is a subDFT of a well-formed DFT.

All notions and notations for DFTs are also used for subDFTs. We emphasise that the subDFTs do not contain any information about basic events or their failure rates.

Like graph rewrite rules, a DFT rewrite rule consists of an interface, left- and right-hand side graphs and two homomorphisms. Additionally, DFT rewrite rules contain a context restriction—defined as a set of DFTs; DFTs which violate the restriction—i.e. DFTs in the set—are not amenable to rewriting by the corresponding rewrite rule. The interface only consists of elements; partitioned into an input (sinks in the graph) and an output interface. We formally define a rewrite rule as follows where we impose some restrictions on the interfaces, which we discuss later on.

Definition 4.4 ((*Structural*) DFT *rewrite rule*) A *DFT rewrite rule* is a tuple $(V_\mathrm{i}, V_\mathrm{o}, L, R, H, \mathfrak{C})$ with

- A set of *input elements* V_i.
- A set of *output elements* V_o with $V_\mathrm{i} \cap V_\mathrm{o} = \emptyset$.
- A well-formed *matching subDFT* $L = (V_L, \sigma_L, Tp_L^\sharp)$.
- A well-formed *result subDFT* $R = (V_R, \sigma_R, Tp_R^\sharp)$.
- Two *graph morphisms* $H = \{h_l, h_r\}$ with
  - an embedding $h_l \colon V_\mathrm{i} \cup V_\mathrm{o} \to L$, and
  - a homomorphism $h_r \colon V_\mathrm{i} \cup V_\mathrm{o} \to R$ such that $h_r \mid V_\mathrm{i}$ is injective.
- A *context restriction* set $\mathfrak{C} = \{(F_i, \zeta_i) \mid i \in \mathbb{N}, F_i \text{ a DFT}, \zeta_i \colon L \nrightarrow V\}$ of *embargo DFTs* where ζ_i is a graph homomorphism for each $i \in \mathbb{N}$,

such that the following conditions hold:

1. The left-hand side elements without children are the input elements: $h_l(V_\mathrm{i}) = \{v \in L \mid \mathsf{child}(v) = \emptyset\}$.
2. No children are added to input elements: $\forall\, v \in h_r(V_\mathrm{i}).\ \mathsf{child}(v) = \emptyset$.
3. All elements in L that are not input elements are typed: $V_L \backslash h_l(V_\mathrm{i}) \subseteq \mathrm{Dom}(Tp_L^\sharp)$.
4. Typing of input elements in L and R agree: $\forall\, v \in V_\mathrm{i}.\ Tp_L^\sharp(h_l(v)) = Tp_R^\sharp(h_r(v))$.
5. Output elements are not typed as FDEPs: $\forall\, v \in V_\mathrm{o}.\ Tp_L^\sharp(h_l(v)) \neq \mathsf{FDEP} \neq Tp_R^\sharp(h_r(v))$.

We abbreviate DFT rewrite rule $(V_\mathrm{i}, V_\mathrm{o}, L, R, H, \mathfrak{C})$ by $(L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \to R, \mathfrak{C})$. By slight abuse of notation, we use h_l (h_r) also to denote the embedding from $V_\mathrm{i} \cup V_\mathrm{o}$ into $\mathcal{L}$ ($\mathcal{R}$).

Example 4.3 Consider the subsumption rule from Example 4.1, where we argued that the rule is not context-free. Recall that Fig. 12 displays the left- and right subDFT where a triangle depicts an untyped element. Let $V_{\text{i}} = \{C, D\}$ and $V_{\text{o}} = \{A\}$. The homomorphism h_l is a trivial embedding (the identifiers share the same name). The homomorphism h_r is given by $A \mapsto A'$, $C \mapsto C'$, $D \mapsto D'$. Furthermore,

$$L = (\{A, B, C, D\}, \{A \mapsto CB, B \mapsto CD, C \mapsto \varepsilon, D \mapsto \varepsilon\}, \{A \mapsto \mathsf{OR}, B \mapsto \mathsf{AND}\})$$

and

$$R = (\{A', C', D'\}, \{A' \mapsto C', C' \mapsto \varepsilon, D' \mapsto \varepsilon\}, \{A' \mapsto \mathsf{OR}\}).$$

Note that we cannot match B on an element with other successors than the element matched by A. If we would allow such match, a dangling edge would result, as we remove the element matched by B.

The context restriction is aimed to ensure that problems as in Example 4.1 cannot occur. For the sake of simplicity, we impose a rather strong restriction; it guarantees that (a) the element matched by D does not have any predecessors after rewriting—by ensuring it has only one predecessor during matching, and (b) the matched element of D is an EE. This means that D is an EE which is independent of other parts after rewriting. Thus, any issues where D is not activated do not play a role here. Formally, the context restriction is given as

$$\mathfrak{C} = \{(F, \zeta) \in \text{DFT} \times (\text{subDFT} \to \text{DFT}) \mid \mathsf{parent}(\zeta(D)) \neq \{\zeta(B)\} \vee Tp(\zeta(D)) \neq \mathsf{EE}\}.$$

It follows that all conditions from Definition 4.4 are fulfilled; this thus defines a DFT rewrite rule.

The semantics of DFT rewrite rules, i.e., (1) when can a rule be applied? and (2) what is result of applying a rule?, are given by a reduction to graph rewriting; cf. Sect. 4.4. We introduce the important notions here on a more intuitive level. A rule can be *matched* on a DFT F if the left subDFT L is a substructure of F. The elements corresponding to L in F are called the matched elements; other elements in F are called *unmatched*. As we want to prevent rules from being applicable in certain contexts, rewrite rules have a possibly empty, possibly infinite, set of DFTs on which they cannot be applied, called *context restrictions*. In these DFTs, we have to mark where the match cannot be applied, which is done by a homomorphism from the left-hand side to the DFT in which the rule cannot be applied. If the DFT which we want to match corresponds to one of the DFTs in the context restriction, we call the DFT from the context restriction an *effective embargo*. In absence of an effective embargo, a *successful match* exists, together with a *matching homomorphism*. The replacement of L by the R in a successful match is called a *rewrite step*.

Remark **2** We usually do not give the context restrictions as an explicit set, but rather by characterising which DFTs should be excluded. The context restriction is then the set of all DFTs which satisfy the characterisation. When operationalising this in GROOVE in Sect. 6, we make use of the advanced features that GROOVE [GdMR+12] has to embed these restrictions into the rewriting directly.

Input- vs. output-interface. To ease the correctness proofs of the DFT rewrite rules, we distinguish an input- and output-interface. Elements in the input- and output-interface are called *input-* and *output-elements*, respectively. Roughly speaking, input-elements are the start of failure propagations through the subDFTs, while output-elements are the endpoints. Intuitively, on rewriting DFT F into F' by replacing subDFT L by R, the elements in the interface are kept untouched—as in standard graph rewriting. In particular, only elements matched by interface vertices in F can have unmatched successors or unmatched predecessors. All successors of output-elements must be matched; this requirement is however not imposed on their predecessors Input-elements are not restricted in the type of connections to unmatched elements. As they are the starting point of failure propagation, we require them to only have unmatched successors; moreover, they should fail as before (based on the failure-behaviour of their children). Thus, changing the type of input elements is forbidden.

Correctness of DFT rules. So far, a rewrite rule is allowed to take any form; in particular it can change the semantics. In Sect. 5.3, we give 29 families of rules. All those rules preserve the reliability and the mean time to failure; that is if F is rewritten to F' by any of the rules, then F and F' give rise to the same reliability and MTTF.

4.4. Graph rewriting for DFTs

In order to formalise the rewriting of DFTs, we reduce this to the well-established area of graph rewriting; this is depicted in Fig. 13. Each DFT has an unique—up to isomorphism—graph representation. Moreover, a graph

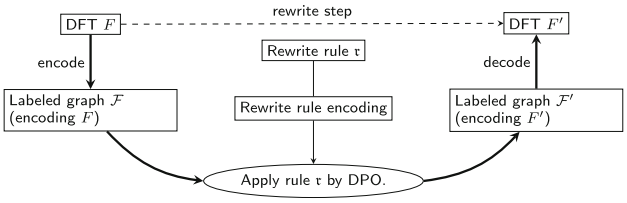


Fig. 13. Formalise DFT rewriting via standard graph rewriting

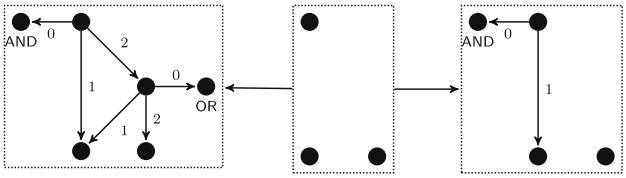


Fig. 14. Graph representation for the DFT subsumption rule in Fig. 12

representation yields an—up to isomorphism—unique DFT. Every DFT rewrite rule can be translated into a DPO graph rewrite rule. A DFT rewrite rule is then matched if the left-hand side graph in the corresponding graph rewrite rule can be matched against the graph representation of the DFT. The effect of the application of a DFT rewrite rule is given by applying the corresponding graph rewrite rule on an encoded DFT. The resulting graph can be decoded, yielding a DFT, which *by definition* is the result of the rewrite step. The remainder of the section formalises this approach.

4.4.1. Encoding structural rewrite rules as graph rewrite rules

Definition 4.5 (*The graph rewrite rule of a* DFT *rewrite rule*) The *corresponding graph rewrite rule* of DFT rewrite rule $(V_\mathfrak{i}, V_\mathfrak{o}, L, R, \{h_l, h_r\}, \mathfrak{C})$ equals $(\mathcal{L}, V_\mathfrak{i} \cup V_\mathfrak{o}, \mathcal{R}, h_l, h_r)$ where $\mathcal{L}$ and $\mathcal{R}$ are the graphs that correspond to DFTs L and R, respectively.

The definition above does not add the carries of the interface elements to the interface of the graph rewrite rule. This is not important, as the carries are only connected to these interface vertices. With this graph rewrite rule, the carries are deleted and added afterwards, thereby not changing them. Note that the attachment function is untouched. Thus, we cannot delete any EEs with attached basic events. Furthermore, the context restrictions are not explicitly reflected in the graph rewrite rule. Graph rewriting has been extended to feature context restrictions (often referred to as negative application conditions), but a translation into these rules is not straightforward. Rather, we check the context restriction on the DFT representation and then apply the rule on the graph representation of a DFT only if the context restriction is met.

Example 4.4 Consider the DFT rewrite rule in Fig. 12. Its graph rewrite rule is given in Fig. 14. The actual homomorphisms between the vertices are given by their relative position in the graph. We see that the two type-less elements do not have carries attached. The graph rewriting removes everything but the tree vertices corresponding to the elements. In particular, it also removes the type of the topmost element. On the right-hand side, this type is added again.

Definition 4.6 (*Match*) Let F be a DFT with graph representation $\mathcal{F}$ and rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$. Injective graph morphism $\kappa : \mathcal{L} \twoheadrightarrow \mathcal{F}$ is a *rule match morphism* if it satisfies:

1. the dangling edge condition (cf. page 14).
2. no successors of output elements are deleted: $\forall\, v \in V_\mathfrak{o}.\ |\ \mathrm{OutV}(v)\ | = |\ \mathrm{OutV}(\kappa(v))\ |$.

For rule match morphism κ, $\kappa(\mathcal{L})$ is called a *matched graph*, $\kappa(V_\mathfrak{i} \cup V_\mathfrak{o})$ the *match glue*, and $\mathfrak{r}$ a *matching rule*.

Note that context restrictions are ignored here; they are captured just below.

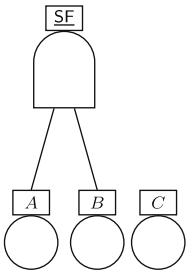


Fig. 15. A small sample DFT with a dispensable EE

Definition 4.7 (*Effective embargo*) Let κ be a rule match morphism on F. The embargo DFT F_i is called *effective* if for some injective homomorphism h from F_i to F it holds:

$$h(\zeta_i(V_{\mathfrak{i}})) = \kappa(h_i(V_{\mathfrak{i}})) \quad \text{and} \quad h(\zeta_i(V_{\mathfrak{o}})) = \kappa(h_o(V_{\mathfrak{o}})).$$

Definition 4.8 (*Successful match*) A matching rule is *successful*, if no $\mathfrak{C}_i$ is an effective embargo.

Definition 4.9 (DFT *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 gr be the corresponding graph rewrite rule of $\mathfrak{r}$. Let L be successfully matched in F by the graph morphism κ. Then F is *rewritten* by $\mathfrak{r}$ using κ to F' if $\mathcal{F} \overset{\mathsf{gr}}{\rightarrow} \mathcal{F}'$ is a graph rewrite step. The tuple $(F, \mathfrak{r}, \kappa, F')$ is a *(DFT) rewrite step*.

The following result asserts that the outcome of a rewrite step again encodes a DFT. We state this result without formal proof; a proof is provided in [Jun15].

Theorem 4.1 *For rewrite step* $(F, \mathfrak{r}, \kappa, F')$*, the graph* $\mathcal{F}'$ *of* DFT F' *is a valid* DFT *encoding.*

4.4.2. Beyond structural rules

We discuss two rewrite steps for removing and merging EEs with attached basic events, respectively. The former reduces the set of basic events the DFT ranges over, the second requires a change of the set of basic events the DFT ranges over.

Removing EE**s.** Event elements that are not connected to the remainder of the DFT can be removed. Removal of EEs generates DFTs which do not range over the same set of basic events as original DFTs, therefore they are outside of our framework of structured DFT rewrite rules presented so far.

Definition 4.10 (*Dispensable* EE*s*) Let $F = (V, \sigma, Tp, \Theta, top)$ be a DFT. An event element $v \in F_{\mathsf{EE}}$ is *dispensable* in F if $\mathsf{parent}(v) = \emptyset$ and $v \neq top$.

Figure 15 shows a simple DFT where C is a dispensable EE. The DFT fails if both A and B have failed.

The elimination aims to remove a EE v from the DFT. Therefore, we would have to make sure that

1. We update all parents to reflect that v is no longer present.
2. We update all incoming FDEPs as they can no longer trigger v.

To address (1), we replace the event element by a $\mathsf{CONST}(\bot)$ element, which in turn may be eliminated by structural rules that also update the parents accordingly. To address (2), we prevent any FDEPs from being present at the first place (notice that $\mathsf{parent}(v) = \emptyset$), and require structural rules to eliminate the FDEPs before eliminating the EE.

Definition 4.11 (*Eliminating a dispensable* EE) Let $F = (V, \sigma, Tp, \Theta, top)$ be a DFT and v be an dispensable EE in F. The *removal* of v in F results in the DFT

$$F' = (V, \sigma, Tp\backslash\{v \mapsto \mathsf{EE}\} \cup \{v \mapsto \mathsf{CONST}(\bot)\}, \Theta \mid (V\backslash\{v\}), top).$$

Evidently, the removal of the dispensable EEs preserves reliability and MTTF of DFTs.

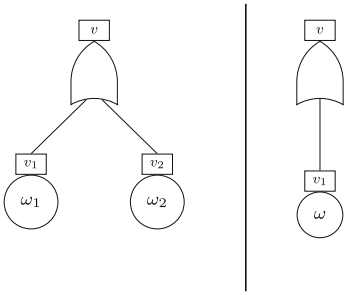


Fig. 16. Merging two EEs with (only) a common OR-gate. Failure rate and dormancy factor of ω are given in Theorem 4.2

Lemma 4.1 *Let F be a* DFT *and v a dispensable* EE *in F. Furthermore, let F' be the result of the dispensable* EE *removal of v. It holds that* $\mathsf{RELY}_F = \mathsf{RELY}_{F'}$ *and* $\mathsf{MTTF}_F = \mathsf{MTTF}_{F'}$.

DFTs where superfluous EEs occur in another fashion, i.e., connected to other elements, can be rewritten into DFTs where elements are disconnected by the use of structural rules. Thus, there is no need to generalise this rule.

Merging EEs. The measures RELY and MTTF are defined independent of the identity of the basic events; therefore, we can also modify these events and preserve RELY and MTTF. The basic idea is depicted in Fig. 16. Two EEs (v_1, v_2) whose attached basic events (ω_1, ω_2) have failure rates of λ_{ω_1} and λ_{ω_2}, respectively, can be merged into a single EE with an attached basic event ω with a failure rate $\lambda_\omega = \lambda_{\omega_1} + \lambda_{\omega_2}$. This is applicable if the EEs are not connected to any other gates. The failure distribution of the OR-gate is given by the minimum over the children. As the children are exponentially distributed, it follows that the OR-gate is also exponentially distributed.

We focus on the binary case, as the n-ary case can be reduced to several instances of the binary case in combination with the structural rewrite Rule 5.

Definition 4.12 (*Independent binary* OR/EE) Let $F = (V, \sigma, Tp, \Theta, top)$ be a DFT and $v \in F_{\mathsf{OR}}$ such that $\mathsf{child}(v) = \{v_1, v_2\} \subseteq F_{\mathsf{EE}}$ and $v_2 \neq top$. Let $\mathsf{parent}(v_1) = \mathsf{parent}(v_2) = \{v\}$. Then (v, v_1, v_2) is an *independent binary* OR/EE *sequence*.

We want to collapse the two EEs into one; and do this by eliminating the second. We update the successor function of the OR-gate: It no longer has two children, instead it has v_1 as only child. We then update the attachment function: v_1 has now ω attached, and notice that v_2 no longer occurs in the DFT.

Definition 4.13 (DFT *obtained by collapsing* OR/EE) Given a DFT F with the independent binary OR/EE sequence (v, v_1, v_2) and $\{\omega, \omega_1, \omega_2\} \subseteq \mathsf{BasicEvents}$ with $\Theta(v_i) = \omega_i$ for $i \in \{1, 2\}$ and $\Theta(x) \neq \omega$ for all $x \in V$. Let $\lambda_{\omega_1}, \alpha_{\omega_1}, \lambda_{\omega_2}, \alpha_{\omega_2}$ be the failure rate and dormancy factor of ω_1, ω_2, respectively. Let ω have failure rate $\lambda_\omega = \lambda_{\omega_1} + \lambda_{\omega_2}$ and dormancy factor $\alpha_\omega = \frac{\alpha_{\omega_1} \lambda_{\omega_1} + \alpha_{\omega_2} \lambda_{\omega_2}}{\lambda_\omega}$. Collapsing (v, v_1, v_2) in F yields the DFT F' with

$$F' = (V \backslash \{v_2\}, \sigma \backslash \{v \mapsto v_1 v_2\} \cup \{v \mapsto v_1\}, Tp|_{V'}, \Theta' \cup \{v_1 \mapsto \omega\}, top)$$

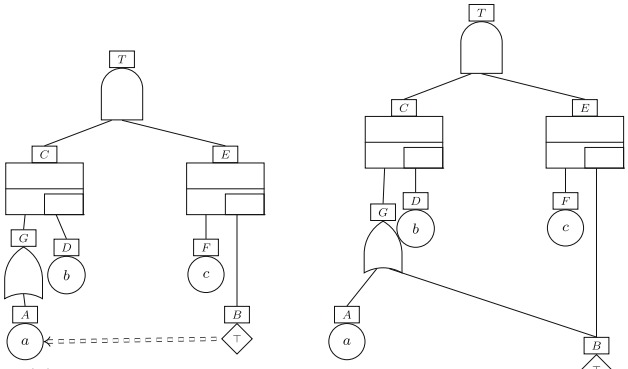
and $\Theta'(v') = \Theta(v')$ for $v' \in V \backslash \{v_1, v_2\}$ and $\Theta'(v_1) = \omega$.

For convenience, we do not remove the OR-gate; OR-gates with single successors can be eliminated by structural rules.

Lemma 4.2 *Given a DFT F with the independent binary* OR/EE *sequence (v, v_1, v_2). Let F' be the DFT obtained from F by collapsing (v, v_1, v_2). Then:* $\mathsf{MTTF}_F = \mathsf{MTTF}_{F'}$ *and* $\mathsf{RELY}_F = \mathsf{RELY}_{F'}$.

The proof of this lemma is rather straightforward and omitted. Note that introducing new EEs with new failure rates goes beyond standard graph rewriting, but is easily integrated in the GROOVE framework (cf. Sect. 6).

The merging rule presented here could be generalised into an additional set of rules, orthogonal to the earlier presented structural rules on DFTs. First of all, we observe that a (deterministic) DFT encodes a phase-type distribution [Neu94]. Obviously, there exist several DFTs which encode the same phase-type distribution, see the lemma above for an example. Each subtrees also encodes a phase-type distribution, given by a DFT which is isomorphic to the subtree. Such a subtree could thus be replaced by another DFT which encodes the same phase-type distribution without affecting the host DFT.



(a) Well-formed DFT. **(b)** Ill-formed DFT.

Fig. 17. Invalid syntax due to merging spare modules

This allows for an automated proof-scheme for replacement of independent subDFTs by other DFTs; using work on the reduction of acyclic phase-type distributions on the CTMC level [PH08].

4.5. Preserving well-formedness

In the last part of this section, we discuss the preservation of well-formedness by the DFT rewrite rules. Applying the rewrite rules on well-formed DFTs may result in a DFT that is not well-formed, as shown by the two examples below. The first example shows that rewriting may introduce dependencies between primary and spare children of SPARE-gates; the second one that rewriting may introduce loops.

Example 4.5 The DFT in Fig. 17a is well-formed. Applying rewrite Rule 24 on page 37 by ignoring the context restrictions leads to the DFT in Fig. 17b. The resulting DFT is ill-formed, since its primary and spare modules are dependent, in particular the semantics for the right-hand side are not defined.

The second rule shows that cyclic DFTs can be constructed. This is possible as acyclicity is a global criterion, while the acyclicity of the host DFT and the rule graphs are local criteria.

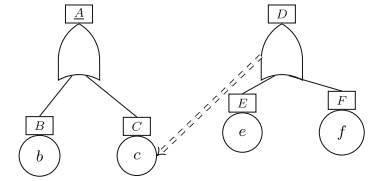
Example 4.6 Consider the DFT from Fig. 18a. Based on the same arguments as in Example 4.5 above, we rewrite the FDEP, and obtain the DFT in Fig. 18b. Now consider the well-formed subDFTs L and R in Fig. 18c and d, respectively. Applying the rule to the DFT in Fig. 18(e) yields the DFT in Fig. 18(f). This DFT is ill-formed due to the introduced cycle.

Thus, we conclude that there are rules with practical relevance that are only applicable on a restricted class of DFTs as they might yield DFTs which are not well-formed. However, for almost all rules, the resulting DFT is well-formed by construction of the rewrite rule; formal criteria are given in [Jun15]. In our prototypical implementation (see Sect. 7), for the two remaining rewrite rules (together encoding Rule 24), we carry out an a-posteriori check. That is, after applying a rewrite step with either of these two rules, it is checked whether the result is a well-formed DFT; if this is not the case, the rule is revoked.

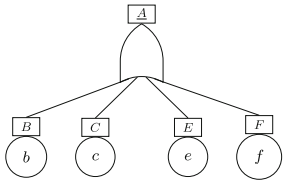
5. A catalogue of rewrite rules

In this section, we present a selection of structural rewrite rules to illustrate the usage of the framework as well as the variety of rules that are possible. We first discuss the existence of normal forms. Then, we discuss a common set of context restrictions. We then first give the rules, and finish the section with a discussion of the correctness proofs for the rules.

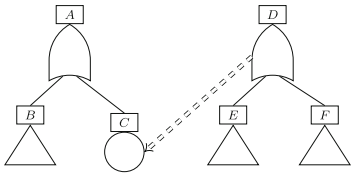
All our rules we present here are ought to be correct in the sense that they preserve our measures-of-interest. Moreover, many rules are symmetric, i.e. they can be applied from left-to-right and from right-to-left. We call a valid rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ *symmetric*, if $(R \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow L, \mathfrak{C})$ is also a valid rewrite rule. We do not want to discuss the application from left-to-right and vice versa separately.



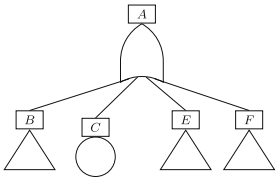
(a) A DFT which can be simplified with a specific rule.



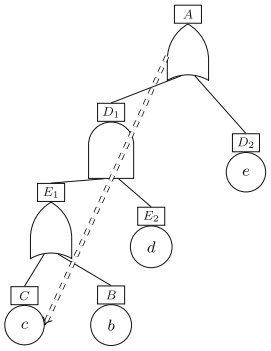
(b) The result after simplification.



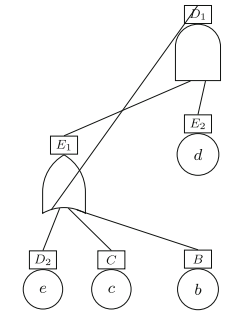
(c) The matching subDFT.



(d) The result subDFT.



(e) Another DFT on which the rule is applicable.



(f) A cyclic and thus ill-formed DFT.

Fig. 18. Introduction of cycles by rewrite rules

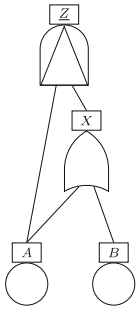
Definition 5.1 (*Valid rewrites*) A rewrite rule $\mathfrak{r}$ is *valid*, if for all DFTs F, F', such that if rewriting F with $\mathfrak{r}$ using κ yields F', it holds that $\mathsf{MTTF}_F = \mathsf{MTTF}_{F'}$ and $\mathsf{RELY}_F = \mathsf{RELY}_{F'}$.

5.1. On the existence of normal forms

One common technique to rewrite complex structures into simpler ones, and to reason about their equivalence, is via normal forms. Static fault trees, as a bounded distributive lattice, can be rewritten by using the lattice axioms. This also leads to well-known normal forms, namely disjunctive and conjunctive normal form. Using the substitution of FDEPs by ORs as outlined in [MRL10], these normal forms also exist in the presence of FDEPs.

For DFTs with PAND and SPARE gates the situation is more complex. The obvious generalisation of disjunctive normal forms (DNFs) to DFTs does not work: To see this, we note that the DNF for an SFT corresponds to enumerating all minimal cut sets, i.e. a minimal set of EEs that makes the SFT fail. Hence, one could try to find a normal form for DFTs by enumerating all minimal cut sequences [RS15], which are the DFT analogon of minimal cut sets. We argue that this is not possible. A DNF has an OR-gate in the top node whose children are all AND-gates and each AND-gate has as children EEs $v_1, v_2, \ldots, v_n$, where $\{v_1, v_2, \ldots, v_n\}$ constitutes a minimal cut set. Generalising this idea to minimal cut sequences means that we replace the AND by a PAND-gate with children, $v_1, v_2, \ldots, v_n$ for each cut sequence $v_1 v_2 \cdots v_n$. This approach, however, fails: Consider the DFT in Fig. 19. Since the OR-gate does not distribute over the PAND-gate, it is not possible to find a fault tree with EEs, AND-, OR-, PAND-gates such that PAND-gates have only EEs as children—this observation is closely related to the lack of information in minimal cut sequences, as described in [JGKS16].

Notice that we did not include any SPARE-gates in the argument; SPARE-gates necessarily add additional complexity—due to well-formedness and activation semantics—to any possible normal form. In fact, we conjecture that a suitable normal form for DFTs is closely related to an enumeration of all failure traces, which is as hard as constructing the underlying state space, which we want to speed-up in the context of this paper. Moreover, due to the syntactical restrictions related to SPARE-gates, it is hard to find a canonical representation for DFTs. Thus, we believe that transforming DFTs into normal forms does not aid our aim of a faster DFT analysis.



π	Minimal cut sequence
ε	✗
A	✓
B	✗
AB	✓
BA	✗

Fig. 19. A sample DFT without a trivial normal form

5.2. Context restrictions

In Sect. 4.1, we have argued the need for context restrictions for DFT rewrite rules. Thus, a context restriction is a subset of $\mathfrak{r}$ indicating when a certain rule cannot be applied; our context restrictions are illustrated in Fig. 20. For readability, some of the context restrictions presented here are stricter than necessary; milder restrictions can be found in [Jun15]. Notice that the restrictions here typically range over (subsets of) the successor- and predecessor closures of elements in the DFT. These sets are finite and any typical set operations are thus also easily computable. For our experimental evaluation, we used the tool GROOVE to implement these checks, but dedicated routines checking the formulated conditions can easily be deduced from the given definitions.

Event-dependent context. Several rules require that a matched subDFT has no EEs that have failed initially, i.e. before any basic event fails. We call elements which can only fail after an EE has failed *event-dependent*. An element is event-dependent if no $\mathsf{CONST}(\top)$ is in the successor-closure; indeed, only elements in the predecessor closure of $\mathsf{CONST}(\top)$ elements can fail initially. A context is event-dependent if it contains no event-dependent elements in its matched subDFT and is illustrated in Fig. 20a.

Definition 5.2 (*Event-dependent context*) Let $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule, and let $x \in V_\mathrm{i}$. Then, the set EventDependentFailure$(x) \subseteq \mathrm{DFTs} \times (L \rightarrow V)$ is defined as

$$\{(F, \zeta) \mid \exists\, y \in \mathsf{dec}_F(\zeta(x))\ y \in \mathsf{CONST}(\top)\}$$

A rewrite rule $\mathfrak{r}$ possesses the *event dependent context restriction w.r.t.* x if

$$\text{EventDependentFailure}(x) \subseteq \mathfrak{C}.$$

Independent input context. The second context restriction is used to ensure that two elements in a matched subDFT never fail due to the same basic event. This means that these elements do not have the same basic event in their successor-closures, see Fig. 20b.

Definition 5.3 (*Independent inputs context*) Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x, y \in V_\mathrm{i}$. The set IndependentInputs$(x, y) \subseteq \mathrm{DFTs} \times (L \rightarrow \text{IndependentInputs}(x, y)_1)$ is defined as

$$\{(F, \zeta) \mid \exists\, z \in F_{\mathsf{EE}}\ z \in \mathsf{dec}_F(\zeta(x)) \wedge z \in \mathsf{dec}_F(\zeta(y))\}$$

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 for traces $\pi \neq \varepsilon$. Together with EventDependentFailures, we can ensure that two elements surely never fail simultaneously.

Activation connection context. Example 4.1 (see Fig. 10 on page 15) has shown that rewriting may change the activation behaviour of elements, and therefore the reliability measures. No such problems occur if either of the following situations hold:

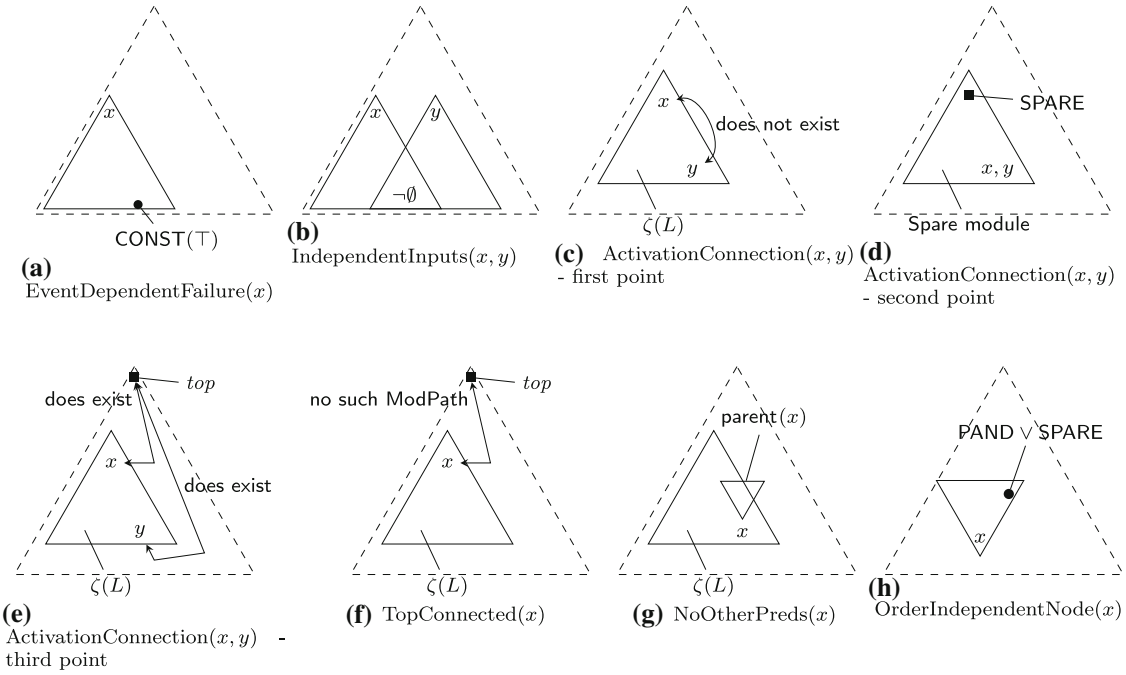


Fig. 20. Context restrictions depicted

1. Elements x and y are in the same module of the DFT and there is an activation connection outside this module, see Fig. 20c;
2. x and y are not in a spare module, and thus never propagate activation, see Fig. 20d; or
3. x and y are not connected (via undirected paths) to the *top* after the application of the rule, see Fig. 20e.

We formalise these properties as predicates ϕ_1, ϕ_2 and ϕ_3 respectively. Since the context restrictions indicate when a rule cannot be applied, this is the case if $\neg(\phi_1(F, \zeta, x, y) \vee \phi_2(F, \zeta, x, y) \vee \phi_3(F, \zeta, x, y))$.

Intuitively, the formula ϕ_1 holds if x and y are connected outside of L; formula ϕ_2 holds if either of the nodes are inside a spare module; and ϕ_3 holds if there is an interface element which remains connected to the element in question even after application of the rule, and that this node is connected to the top, without going through the left-hand side of the rule. Notice that this last addition is essential, otherwise the formula is satisfied even if the node is disconnected by the rule application. Great care has to be taken here to ensure that the node is really connected afterwards—if a path fragment removed by L is not added in R again, then the nodes are disconnected.

Definition 5.4 (*Activation connection context*) Let $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x, y \in V_\mathfrak{i} \cup V_\mathfrak{o}$. The set $\mathrm{ActivationConnection}(x, y) \subseteq \mathrm{DFTs} \times (L \rightarrow \mathrm{ActivationConnection}(x, y)_1)$ 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

$$\begin{aligned}
\phi_1(F, \zeta, x, y) &= \forall\ p \in \mathrm{ModPath}_F(\zeta(x), \zeta(y))\ p = v_0 e_1 v_1 \ldots e_n v_n \exists\, 1 \leq i \leq n\ e_i \in \zeta(L) \\
\phi_2(F, \zeta, x, y) &= \phi_2'(F, \zeta, x) \vee \phi_2'(F, \zeta, y) \\
\phi_2'(F, \zeta, z) &= \exists\, r \in \mathsf{SMR}_F \mathrm{ModPath}_F(\zeta(z), r) \neq \emptyset \\
\phi_3(F, \zeta, x, y) &= \phi_3'(F, \zeta, x) \vee \phi_3'(F, \zeta, y) \\
\phi_3'(F, \zeta, z) &= \exists\, z' \in (V_\mathfrak{i} \cup V_\mathfrak{o}) \exists\, \mathrm{Path}_R(z, z') \neq \emptyset \wedge \exists\, p' \in \mathrm{Path}_F(\zeta(z'), \mathit{top}) \text{ s.t.} \\
&\quad p' = v_0 e_1 \ldots v_n \quad \forall\ i < j\ v_i, v_j \in \zeta(V_\mathfrak{i} \cup V_\mathfrak{o}) \\
&\qquad v_i \ldots v_j \in \mathrm{Path}_{\zeta(L)}(v_i, v_j) \Rightarrow \mathrm{Path}_R(v_i, v_j) \neq \emptyset.
\end{aligned}$$

A rewrite rule $\mathfrak{r}$ possesses the *activation connection context restriction w.r.t. x and y* if $\text{ActivationConnection}(x, y) \subseteq \mathfrak{C}$.

Top module connection context. This rule is similar to the activation connection context. Whereas activation connection context aims at removing connections, this restriction concerns connecting elements. This is a situation which is less common, since the application of such rules is already severely restricted by wellformedness-criteria. Here, we want to enforce that a node is in the top module before applying the rule, see Fig. 20f.

Definition 5.5 (*Top connected context*) Let $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ be a rewrite rule with $x \in V_\mathfrak{i} \cup V_\mathfrak{o}$. The set $\text{TopConnected}(x) \subseteq \text{DFTs} \times (L \rightarrow \text{TopConnected}(x)_1)$ is defined as

$$\{(F, \zeta) \mid \text{ModPath}_F(\zeta(x), top) = \emptyset\}$$

A rewrite rule $\mathfrak{r}$ possesses the *activation connection context restriction w.r.t. x* if $\text{TopConnected}(x, y) \subseteq \mathfrak{C}$.

No other predecessor context. Recall that only elements in the interface are allowed to have predecessors outside of the matched subDFT. Output elements typically have these predecessors. Input elements typically either have successors, or are event elements with attached basic events. In some cases, it is helpful to disallow matching elements with unmatched predecessors, see Fig. 20g.

Definition 5.6 (*No other predecessor context*) Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ with $x \in V_\mathfrak{i}$. The set $\text{NoOtherPreds}(x) \subseteq \text{DFTs} \times (L \rightarrow \text{NoOtherPreds}(x)_1)$ is defined as

$$\{(F, \zeta) \mid \mathsf{parent}_F(\zeta(x)) \backslash \zeta(L) \neq \emptyset\}$$

A rewrite rule $\mathfrak{r}$ possesses the *no-other-predecessor context restriction w.r.t. x* if $\text{NoOtherPreds}(x) \subseteq \mathfrak{C}$.

Order-independent node context. Some rules are correct within static fault trees—especially regarding functional dependencies many simplifications are possible in such a static context. Indeed, if the dependent event of a functional dependency cannot influence the state of dynamic gates, then we can assume that the dependent event fails simultaneously with the trigger. The order-independent node context restricts rule applications by preventing nodes which directly (that is, not via an additional functional dependency) influence dynamic gates. We realise this by prohibiting any SPARE-gate or PAND-gate in the predecessor-closure of x, see Fig. 20h.

Definition 5.7 (*Order-independent node context*) Given a rewrite rule $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ with $x \in L$ The set $\text{OrderIndependentNode}(x) \subseteq \text{DFTs} \times (L \rightarrow \text{OrderIndependentNode}(x)_1)$ is defined as

$$\{(F, \zeta) \mid \mathsf{anc}_F(\zeta(x)) \cap (F_{\mathsf{SPARE}} \cup F_{\mathsf{PAND}}) \neq \emptyset\}$$

A rewrite rule $\mathfrak{r}$ possesses the *order-independent node context restriction w.r.t. x* if $\text{OrderIndependentNode}(x) \subseteq \mathfrak{C}$.

5.3. Structural rules

Notation. The rules presented are potentially infinite families of rules. To obtain a concrete rule from a family, concrete values for every variable occurring have to be selected and all node types (except for the input nodes) need to be instantiated. This value and type instantiation has to conform to the given restrictions (if any). Some input nodes are marked as optional, which means that the rule is applicable both with and without these nodes. Evidently, optional nodes need to be consistently removed (or considered) at both the left-hand as right-hand side of the rule. We adhere to the following conventions:

- The subDFT L is depicted on the left-hand side, while the subDFT R is depicted on the right-hand side.
- All graph elements are uniquely labelled. These labels are not part of the formal definition but are convenient to formulate proofs and indicate homomorphisms.
- Type-less elements are depicted by an upwards-pointing triangle, as used in [SVD+02] 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.

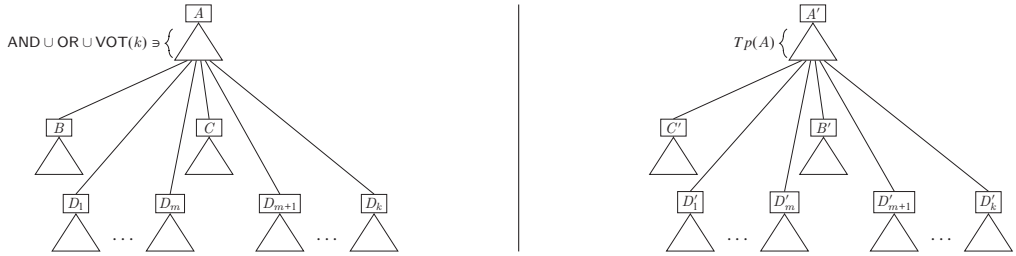
- Input and output elements are given by stating their identifiers in L. We then map them to the identifiers in R to display the homomorphism from $V_{\mathsf{i}} \cup V_{\mathsf{o}}$ to R.
- Any restrictions on variable concretisation or type instantiation, as well as optional nodes (denoted opt.), are depicted with curly braces in the representation of L.

Organisation of the rules. 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, and include the constant elements with rules derived from bounded lattice axioms. We continue with and voting gates, then consider simple rules with PANDs as a tool to describe sequences, and continue with the PAND with different successor eleme nts. We finally present some elimination rules for FDEPs.

5.3.1. Structural identities

Commutativity. The static gates AND, OR, and VOT(k) are commutative. This is presented by a single family of rules in which the subDFTs labeled B and C are swapped.

Rewrite rule 1 Commutativity of static gates.



Input: $\{D_i \mapsto D'_i\}_{i=1}^{k} \cup \{B \mapsto B', C \mapsto C'\}$
Output: $\{A \mapsto A'\}$

Functional congruence. 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. Stated 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: $\{A_1 \mapsto A'_1, A_2 \mapsto A'_1\}$

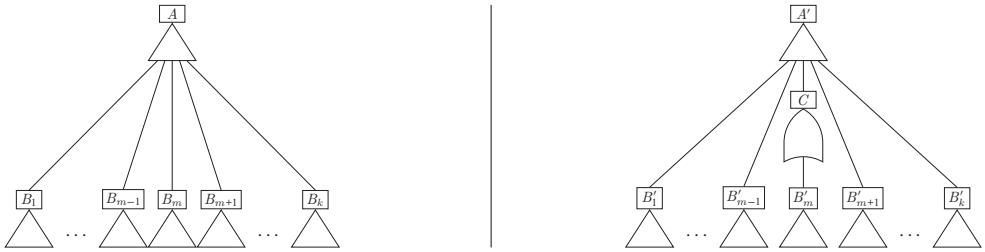
Gate with a single successor. Gates with just a single child fail together with this child, so they can directly be eliminated. Our framework does not allow this rule to be applied from right to left, as information on the type of A is lacking.

Rewrite rule 3 Gate with a single successor.

Input: $\{B \mapsto B'\}$
Output: $\{A \mapsto B'\}$

Adding superfluous gates. To keep other rules (especially Rule 24) more general, it is beneficial to define a rule replacing a successor with an OR-gate. This rule can be generalised to other static gates; e.g., the added OR-gate can be turned into a voting gate, subsequently into an AND-gate, and finally as PAND-gate.

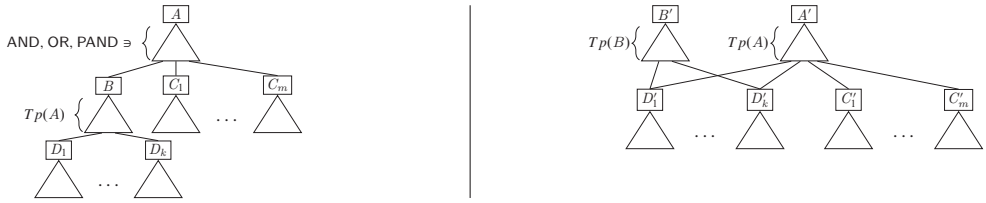
Rewrite rule 4 Add an OR in between.



Input: $\{B_i \mapsto B'\}_{i=1}^{k}$
Output: $\{A \mapsto A'\}$

Left-flattening. Elements whose first successor is of the same type can be merged with that successor. The rule asserts that if A's first successor B has the same type as A, the DFT can be flattened such that B's successors become A's successors (instead of its grandchildren). Its correctness in case A is an AND- or OR-gate is obvious; if A is a PAND-gate, the correctness follows from the fact that the ordering of B's successors is maintained. 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, the restriction to the first successor is essential. Notice that we keep B; thereby the rule is more generally applicable. Using Rule 6 defined below, we can eliminate B in many cases, cf. Example 5.1.

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: $\{A \mapsto A', B \mapsto B'\}$

Elements without predecessors If an element does not have a predecessor, its failure is never propagated.

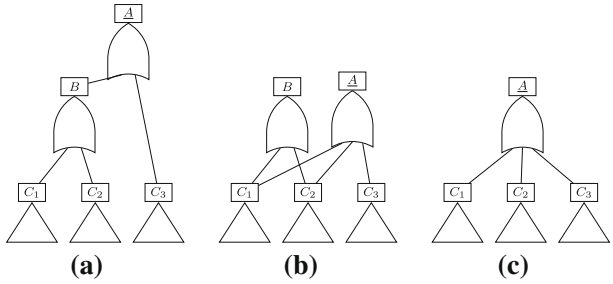
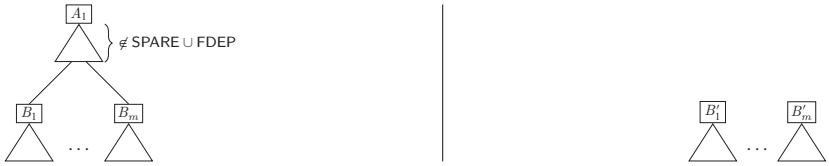


Fig. 21. Flattening and deflattening of DFTs

Rewrite rule 6 Remove elements without a predecessor.



Input: $\{B_i \mapsto B_i'\}_{i=1}^m$
Output: $\emptyset$
Context Restriction: $\{\text{ActivationConnection}(B_i, B_j) \mid 1 \leq i < j \leq m\}$.

The node A_1 is not part of the interface, therefore it cannot have any predecessors in the matched graph. We can thus remove A_1 without affecting failure propagation. The element may, however, connect two otherwise unconnected subgraphs, thereby constructing a single module. Thus, the rule can only be applied in restricted contexts where this activation connection is not important. This is expressed by the context restriction.

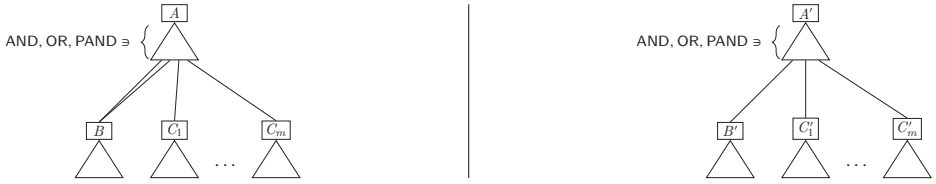
In given DFTs, all elements typically have predecessors. However, after the application of, e.g. Rule 5, this often changes, as illustrated in the next example.

Example 5.1 Consider the DFT given in Fig. 21a. By applying Rule 5, we obtain the DFT in Fig. 21b. Now, B is neither the top-element nor does it have predecessors, so it can be eliminated by Rule 6.

We also refer to the combination of these rules as *flattening* (left-to-right) and *deflattening* (right-to-left).

Equal successors. In the course of rewriting, a gate may have multiple equal successors. For AND- and OR-gates, we can safely remove one of them. For PAND-gates, this can be done if the two equals successors are direct neighbours. The rule below eliminates the first successor of a gate if it is identical to the second one. Due to commutativity, this yields a general rule for AND and OR. For PAND-gates, this suffices when used in combination with rewrite Rule 20.

Rewrite rule 7 Identical leftmost successors of AND, OR or PAND.



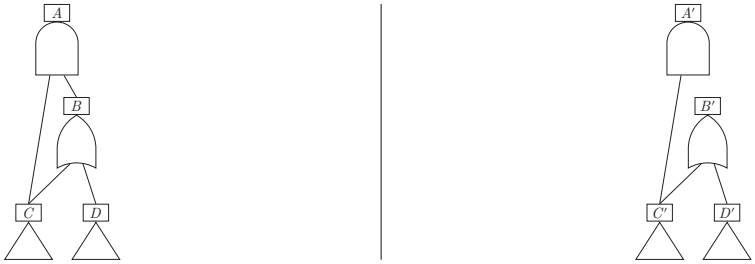
Input: $\{C_i \mapsto C_i'\}_{i=1}^m \cup \{B \mapsto B'\}$
Output: $\{A \mapsto A'\}$

5.3.2. Rules originating from lattice axioms

Simple lattice axioms such as commutativity (Rule 1) and associativity for n-ary gates (Rule 5) have already been treated. Below we consider notions such as subsumption, distributivity, and elimination of constants.

Subsumption. The following subsumption rules are slightly more general than those in Example 4.1. The first subsumption rule subsumes an OR by an AND-gate. Note that B may have predecessors. In case it has predecessors, only the connection is removed, thereby simplifying A. But B is not removed. In absence of predecessors, B can be eliminated by rewrite Rule 6.

Rewrite rule 8 Subsumption of OR-gates by AND-gates.



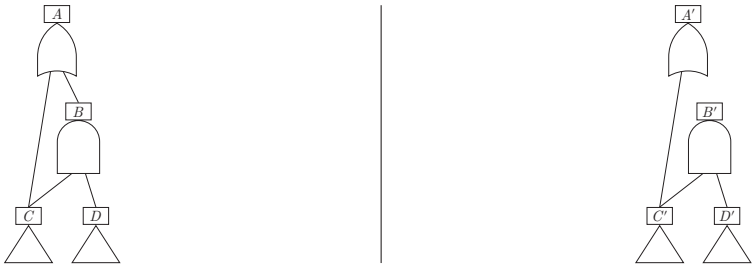
Input: $\{C \to C', D \to D'\}$
Output: $\{A \to A', B \to B'\}$

This subsumption rule is presented for the binary case in order to keep the presentation and the proof obligation simple. Note that this is not a restriction: using flattening, we can apply the subsumption rules also to n-ary gates. This is illustrated by the following example.

Example 5.2 Consider the extension of the DFT from Example 4.1 given in Fig. 22a. As we intend to apply subsumption, we first turn the gates A and B into binary ones. In order to do so, we want to apply deflattening (rewrite Rule 5). As we only defined left-(de)flattening, we apply commutativity (Rule 1) to B to reorder its successors (Fig. 22b). We can now apply deflattening (in reverse direction) on A and B, obtaining Fig. 22c. Reordering B's successors (Rule 1) yields the DFT in Fig. 22d. We can now apply subsumption, yielding Fig. 22e. We restore the successors of the AND-gate by flattening (Rule 5) on Z and A. This yields Fig. 22f. By Rule 5, we also flatten B and Z'. We then remove B by Rule 6) as it has no predecessors (and the context restriction is met). This yields Fig. 22g. By Theorem 4.1, the EEs D and X can be removed, resulting in Fig. 22h.

The subsumption rule for OR-gates over AND-gates is analogous to the previous rule.

Rewrite rule 9 Subsumption of AND-gates by OR-gates.



Input: $\{C \to C', D \to D'\}$
Output: $\{A \to A', B \to B'\}$

Distribution. The distribution rule directly originates from the lattice axioms. As for subsumption, we present the rule for binary gates. Using (de)flattening, we can apply a similar rewriting to n-ary gates.

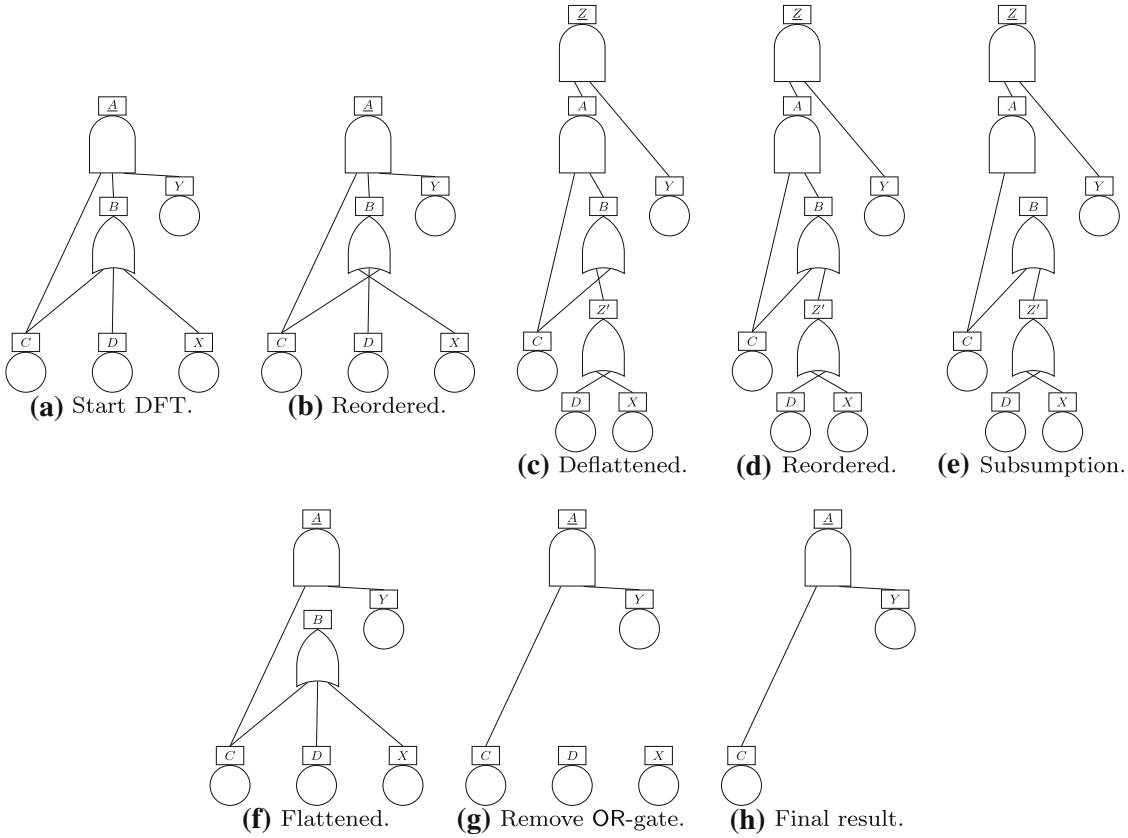
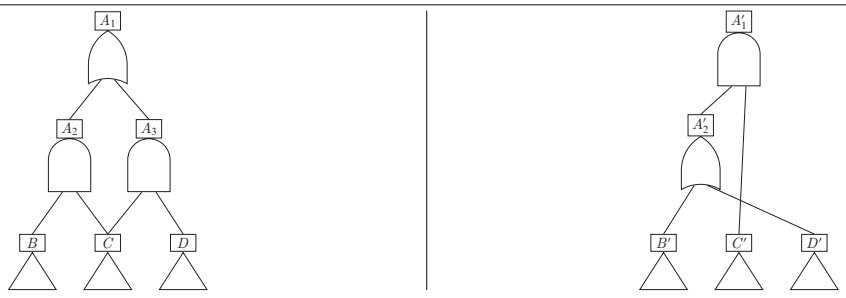


Fig. 22. Applying subsumption in a larger DFT as explained in Example 5.2

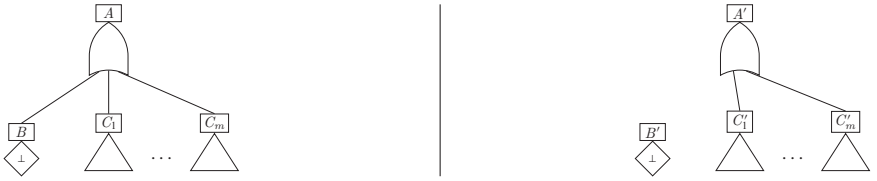
Rewrite rule 10 Distributing OR-gates over AND-gates.



Input: $\{B \mapsto B', C \mapsto C', D \mapsto D'\}$
Output: $\{A_1 \mapsto A'_1\}$

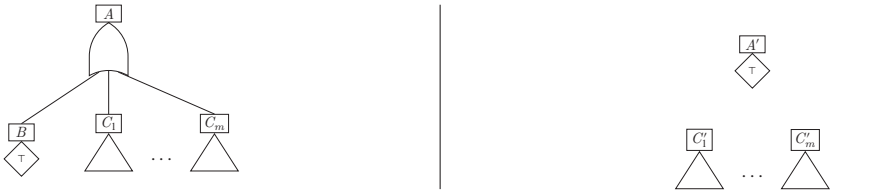
Constant elimination. We now present some rules to eliminate constant elements. Either a constant successor can be removed, as it does not affect the gate (Rule 11 below) or the constant element is propagated upwards as it determines the outcome of its predecessor gate (Rule 12 below). As some connections are removed by these rules, context restrictions are imposed to ensure that the activation context remains intact. Recall that $\bot$ stands for fail-safe elements, while $\top$ denotes an element that has already failed.

Rewrite rule 11 OR-gates with fail-safe successors.



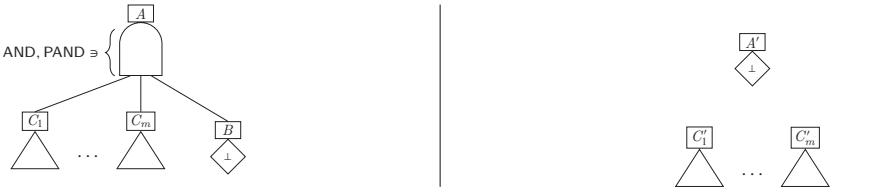
Input: $\{C_i \mapsto C'_i\}_{i=1}^m$
Output: $\{A \mapsto A', B \mapsto B'\}$
Context Restriction: {ActivationConnection(A, B)}

Rewrite rule 12 OR-gates with already failed successors.



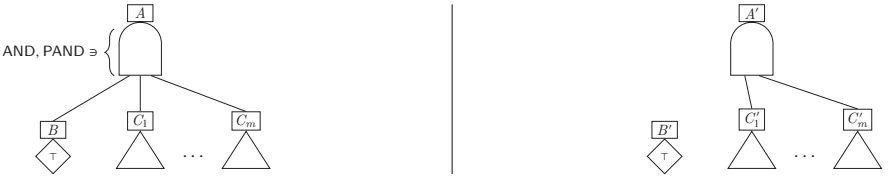
Input: $\{C_i \mapsto C'_i\}_{i=1}^m$
Output: $\{A \mapsto A', B \mapsto A'\}$
Context Restriction: {ActivationConnection(A, C_i) | $1 \leq i \leq m$}

Rewrite rule 13 AND-gate with a fail-safe successor.



Input: $\{C_i \mapsto C'_i\}_{i=1}^m$
Output: $\{A \mapsto A', B \mapsto A'\}$
Context Restriction: {ActivationConnection(A, C_i) | $1 \leq i \leq m$}

Rewrite rule 14 AND-gate with an already failed element as successor.



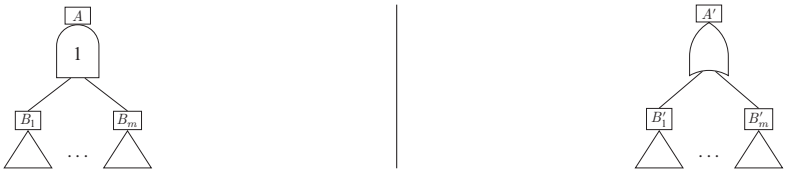
Input: $\{C_i \mapsto C'_i\}_{i=1}^m$
Output: $\{A \mapsto A', B \mapsto B'\}$
Context Restriction: {ActivationConnection(A, B)}

5.3.3. Rules for voting gates

AND- and OR-gates are both special voting gates. Reversely, voting gates can be modelled by OR- and AND-gates. This is reflected in the following two rules.

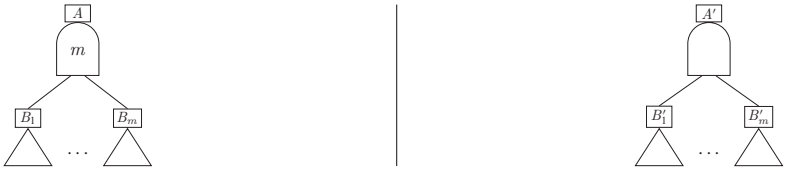
Simple rules for voting gates. The translation of voting gates into AND- and OR-gates is rather trivial. Note that these rules allow any OR-gate with one child to be rewritten into an AND-gate with one child.

Rewrite rule 15 VOT(1) is an OR-gate.



Input: $\{B_i \mapsto B'_i\}_{i=1}^{m}$
Output: $\{A \mapsto A'\}$

Rewrite rule 16 VOT(m) with m successors is an AND-gate.



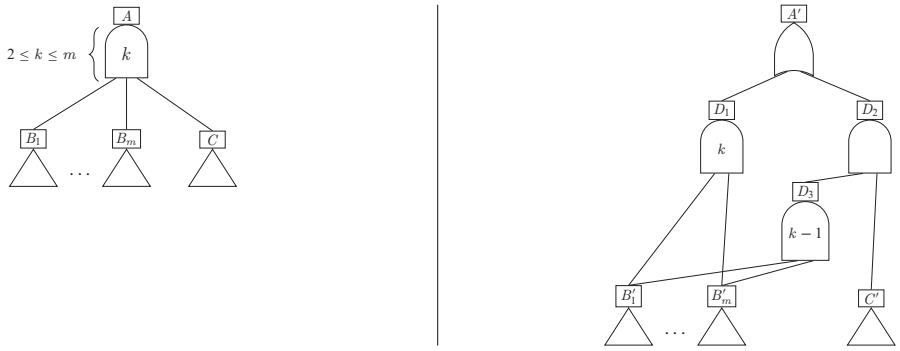
Input: $\{B_i \mapsto B'_i\}_{i=1}^{m}$
Output: $\{A \mapsto A'\}$

Shannon expansion For the expansion of voting gates, we exploit Shannon expansion. Shannon expansion is based on the following proposition for Boolean function f:

$$f(x, y_1, \ldots, y_m) = (x \wedge f(1, y_1, \ldots, y_m)) \vee (\neg x \wedge f(0, y_1, \ldots, y_m)).$$

In the rule presented below, the successor C corresponds to the variable x in the function f. If C fails, then gate A—representing $f(x, y_1, \ldots, y_m)$—fails when $k-1$ out of the m other successors failed. Moreover, regardless of C's failure, A fails when k out of the remaining m successors fail.

Rewrite rule 17 Shannon expansion for VOT(k), $k > 1$.



Input: $\{B_i \mapsto B'_i\}_{i=1}^{m} \cup \{C \mapsto C'\}$
Output: $\{A \mapsto A'\}$

Figure 23 gives an example that shows how the rules presented so far can be used to eliminate fail-safe successors of voting gates.

5.3.4. Conflicting and combined sequences

The next set of rewrite rules focus on PAND-gates. The rules exploit the order requirements of PAND-gates.

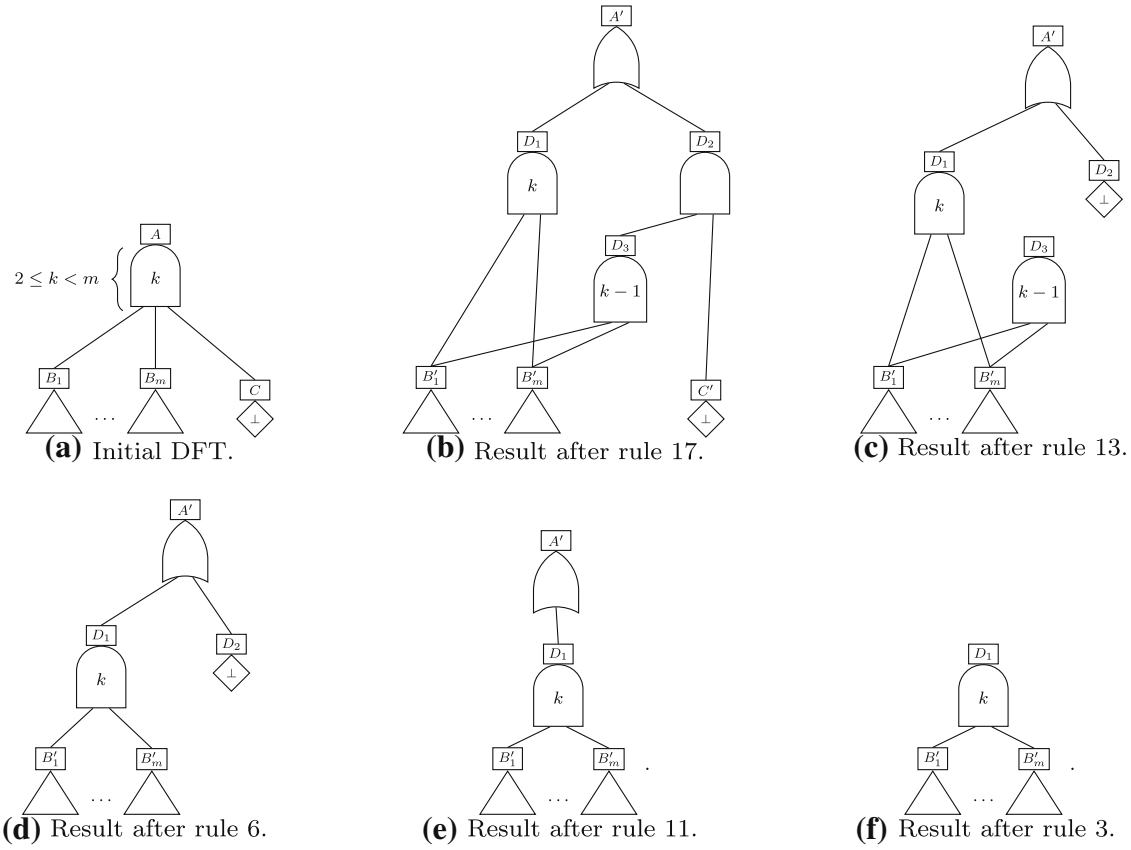
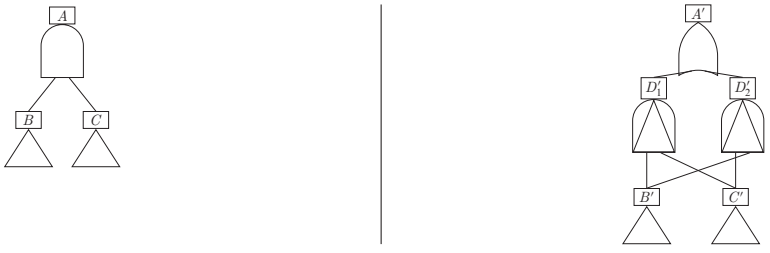


Fig. 23. Steps for rewriting $\mathsf{VOT}(k)$ with an infallible element

Combined sequences. An AND is a PAND-gate without ordering requirement. This can be expressed as allowing all possible failing orderings using an OR-gate (see rule below). Note that once the rule is applied, PAND-gates can possibly be combined with other PAND-gates. Applying from right-to-left, the rule can be used to eliminate PAND-gates.

Rewrite rule 18 Representing PAND-gate using OR- and PAND-gates.

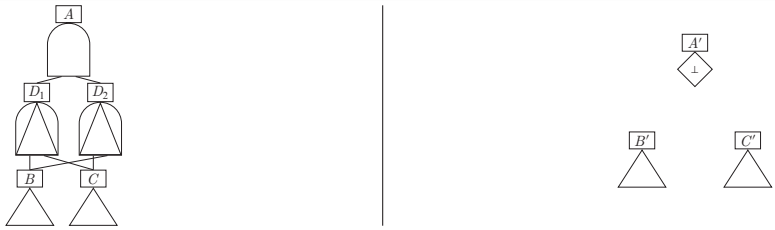


Input: $\{B \mapsto B', C \mapsto C'\}$
Output: $\{A \mapsto A'\}$

Conflicting PANDs**.** PANDs with mutually exclusive ordering requirements can be simplified considerably. Consider a DFT rooted by an AND-gate with two PAND-successors which have conflicting ordering requirements: D_1 requires B to fail before C, whereas D_2 requires the opposite (see the rule below). Under the assumption that the successors never fail simultaneously (ensured as B and C having independent inputs), the PAND-gates D_1 and D_2 can never both fail. Therefore, the AND-root can never fail, as indicated in the DFT on the right.

This simplification however is incorrect if both successors are constantly failed. The context restriction therefore imposes that at most one successor is connected to a constant failure. This assumption may be too strong on its own: If the constant failure does not propagate to a successor, then the PAND-gate would still be conflicting. Other rules, however, ensure that in this case, the CONST(⊤) is eventually eliminated, after which this rule becomes applicable.

Rewrite rule 19 Conflicting PAND-gates with independent successors.



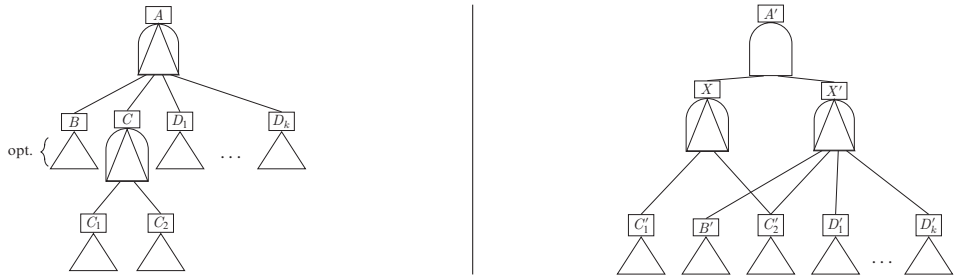
Input: $\{B \to B', C \to C'\}$
Output: $\{A \to A'\}$
Context Restriction: IndependentInputs(B, C)
EventDependentFailure(B)
ActivationConnection({(A, B), (A, C)})

5.3.5. PAND*s with various gates as successors*

We consider simplification of PAND-gates with an arbitrary successor. This is often helpful to find possible conflicts and order-independencies and to successively eliminate PAND by the rewrite rules presented before.

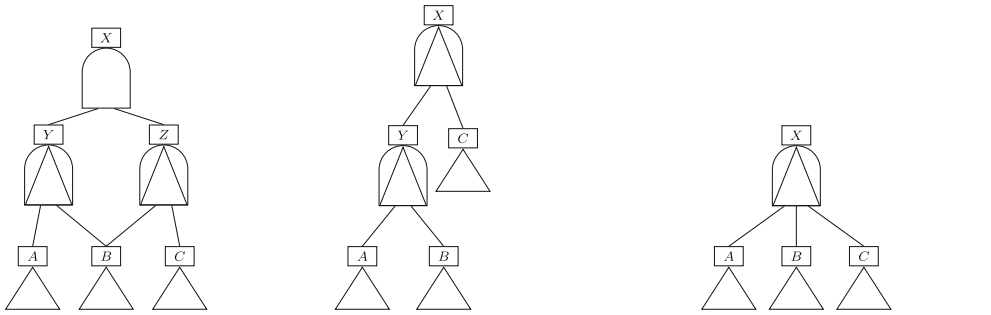
PAND **and** PAND**.** As we observed before, static gates with successors of the same type can be merged. Due to commutativity, this is independent of their position. For PAND-gates, this does not hold. However, PAND-gates with a PAND-successor can be rewritten independent of the successor's position. The corresponding rule (see below) considers a PAND-gate whose second successor is a PAND. Arbitrary positions of the PAND-successor can be handled by using the left-flattening (reverse) of the PAND-gate so as to group all successors (into a single new PAND-gate) that are left from the PAND-gate subject to the rule application. The ordering of C_1 and C_2 (see lhs) is ensured by X (rhs) whereas the fact that B must fail prior to C_1 and C_2 (in that order) is guaranteed by the PAND-gate X' (rhs). This PAND-gate also ensures that B and C_2 should fail before D_1–D_k fail. (Note that successor B is optional.) We emphasize that the rule from right-to-left has great practical relevance. It allows, together with deflattening, to rewrite PAND-chains, as found in, e.g. the sensor-filter case study (cf. Sect. 7). This is illustrated in Fig. 24.

Rewrite rule 20 PAND-gate with a PAND-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: $\{A \mapsto A'\}$

PAND **and first** OR**.** The next rule simplifies a PAND-gate whose first successor is an OR-gate (as argued in Sect. 5.1, there is no general rule that removes OR-gates at arbitrary successor positions of a PAND-gate.) A PAND whose first successor C is a binary OR-gate with successors C_1 and C_2 can be rewritten into a disjunction

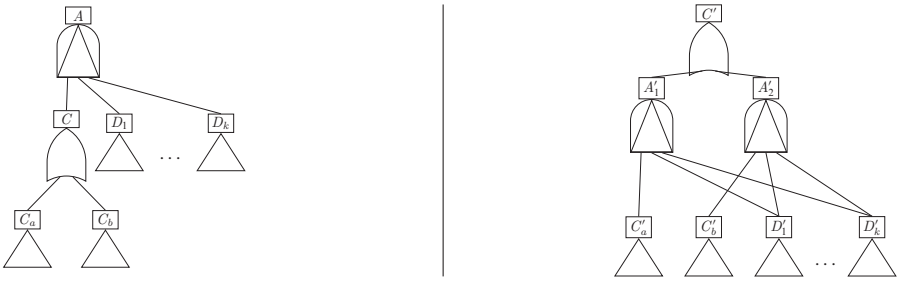


(a) A sample DFT. **(b)** Result after rule 20. **(c)** Result after flattening and cleaning.

Fig. 24. Rewriting a chain of PANDs

of two PAND-gates where the first one takes care of the failure of C_1 whereas the second PAND treats the failure of C_2. Notice that by (de-)flattening, this rule can be applied to an n-ary PAND-gate.

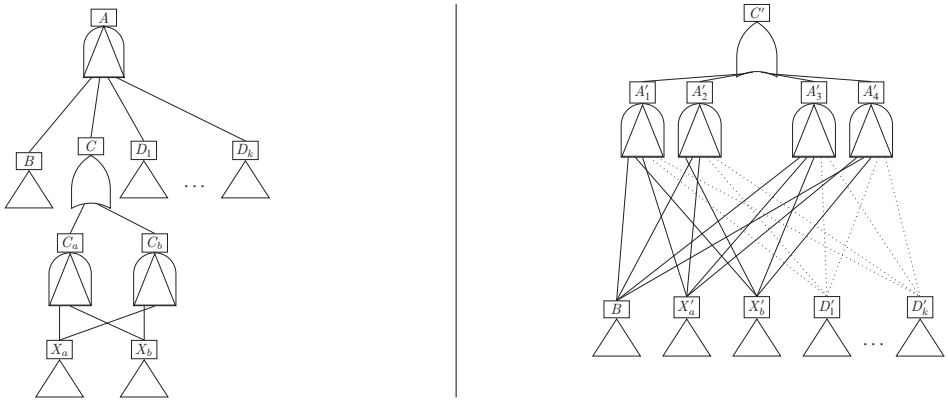
Rewrite rule 21 PAND-gate with a first OR-successor.



Input: $\{C_a \mapsto C'_a, C_b \mapsto C'_b\} \cup \{D_i \mapsto D'_i\}_{i=1}^{k}$
Output: $\{A \mapsto C'\}$

PAND **and special** OR **&** PAND **and** AND**.** For several special cases, there are rules to eliminate OR-gates that are not the first successor of PAND-gates. As an example, we present a rule, which together with Rule 18 allows rewriting of AND-gates that are successors of an PAND-gate.

Rewrite rule 22 PAND-gate with an OR-successor (special case).

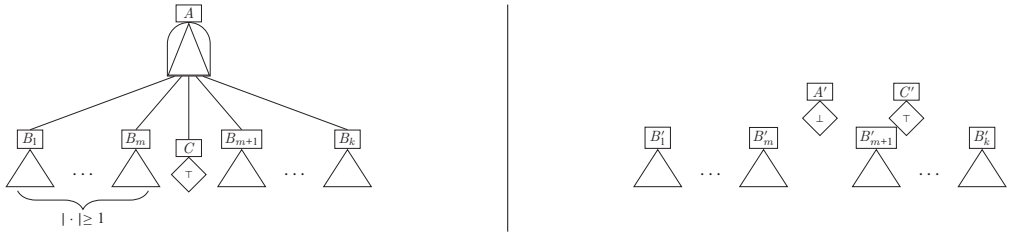


Input: $\{B \mapsto B', X_a \mapsto X'_a, X_b \mapsto X'_b\} \cup \{D_i \mapsto D'_i\}_{i=1}^{k}$
Output: $\{A \mapsto C'\}$

Note that the DFT rooted at C (left-hand side) corresponds to the right-hand side of Rule 18. Note that C fails if X_a and X_b fail in arbitrary order. On the right-hand side, the dotted part indicates that $D_1 \dots D_k$ fail in order. The two possible failing orders of X_a and X_b give rise to the two pairs of PANDs A'_1, A'_2 and A'_3, A'_4, respectively. Each pair contains two variants: either B fails before X_a or X_b do, or it does so after X_a or X_b fails.

PAND **and** CONST(⊤)**.** Complementary to the rules for eliminating constant elements as first (last) successor of a PAND (cf. Rules 13 and 14), we now consider a rule for successors of a PAND that already failed. It expresses that the PAND can never fail if the left neighbours of the failed successor have not failed at initialisation. We ensure this via a context restriction on the event dependencies.

Rewrite rule 23 PAND-gate with CONST(⊤) as non-first successor.

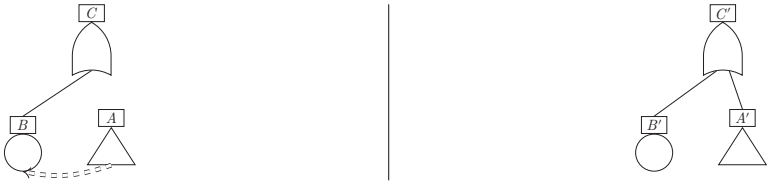


Input: $\{B_i \mapsto B'_i\}_{i=1}^{k}$
Output: $\{A \mapsto A', C \mapsto C'\}$
Context Restriction:
$\{\text{ActivationConnection}(X, B_i) \mid 1 \leq i \leq k \wedge X \in \{A, C\}\}$
$\{\text{ActivationConnection}(B_i, B_j) \mid 1 \leq i < j \leq k\}$
$\{\text{ActivationConnection}(A, C)\}$
$\{\text{EventDependentFailure}(B_i) \mid 1 \leq i \leq m\}$

5.3.6. Representing FDEP*s via* OR*-gates*

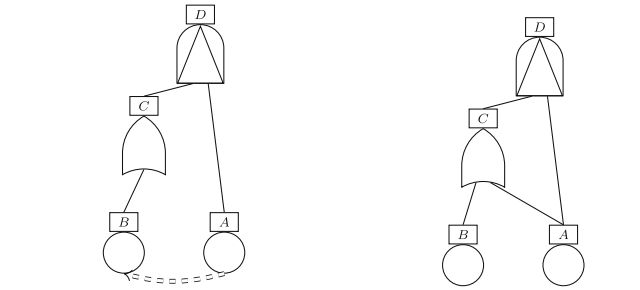
As mentioned earlier, functional dependencies are syntactic sugar for static fault trees. The following rule, adopted from [MRL10], rewrites a functional-dependency into an OR-gate. Due our simplified interfaces of rewrite rules, the rule is slightly more involved than one might expect; this is reflected by the OR-gate with a single successor. Note that this OR-gate with one successor can be easily constructed by applying Rule 4.

Rewrite rule 24 Eliminating FDEPs by an OR-gate.



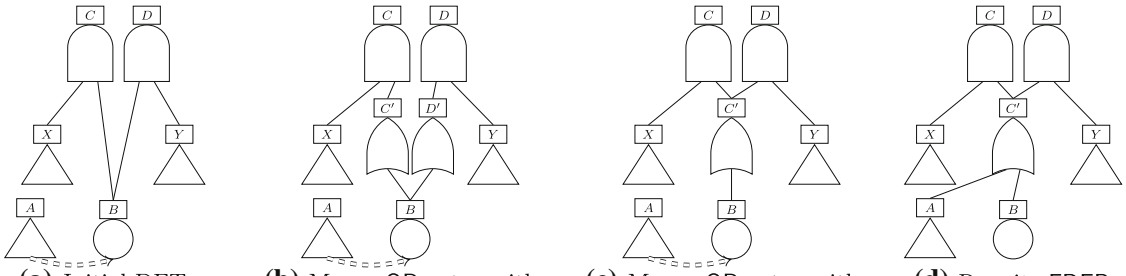
Input: $\{A \mapsto A', B \mapsto B'\}$
Output: $\{C \mapsto C'\}$
Context Restriction:
NoOtherPreds(B)
OrderIndependentNode(B)
TopConnected(B)
TopConnected(C)

Let us explain the context restrictions of this rule. Eliminating FDEPs has some intricate consequences. First, B no longer fails once A does. Any element connected to B should thus be connected via C (which still fails).



(a) D does not fail with A **(b)** D does fail with A

Fig. 25. Order-dependencies in FDEP make rewriting (**a**) into (**b**) is not valid



(a) Initial DFT. **(b)** Merge OR-gates with equal successors. **(c)** Merge OR-gates with equal successors. **(d)** Rewrite FDEP.

Fig. 26. Circumventing the NoOtherPreds(B) context restriction in Rule 24

This is ensured by the NoOtherPreds context restriction on B. Second, C fails strictly after A. However, after rewriting A and C fail simultaneously. This might lead to several changes in behaviour in the DFT. The simplest issue is depicted in Fig. 25. In Fig. 25a, a failure of (only) A makes D infallible, as the ordering requirement of D is violated. It then causes the failure of B, which is propagated but does not influence D anymore. Rewriting this FDEP according to our rule yields Fig. 25b. Here, the failure of A is propagated to C immediately, causing all children of D to fail simultaneously, which means that D fails as well. To prevent this, we impose the context restriction OrderIndependentNode on B.

Third, the rule may extend a spare module by connecting elements previously outside the spare module. Notice that it cannot join two spare modules, as this violate well-formedness (cf. Sect. 4.5). To ensure that this extra connection does not cause any additional activations, we require both A and C to be in the same spare module as the *top*, which means that there is no joining of spare modules. This is ensured by the context restrictions TopConnected.

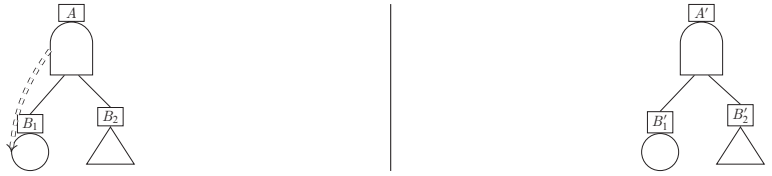
Note that requiring B to have no other predecessors is not restrictive, as DFTs amenable to FDEP elimination can be transformed into this form. This is illustrated in Fig. 26.

5.3.7. Additional rules for FDEPs

We present some other rules for eliminating dispensable FDEPs. In the remainder of this section, we do not aim for completeness, as the number of rules would become too large. The presented rules indicate possible transformations that are not always captured yet by general rules such as Rule 24.

Triggers never fail before dependent events. If the FDEP is triggered after the failure of the dependent event, it does not influence anything else. If A fails, then surely B_1 has failed. This means that we cannot let B_1 fail anymore, the FDEP is superfluous.

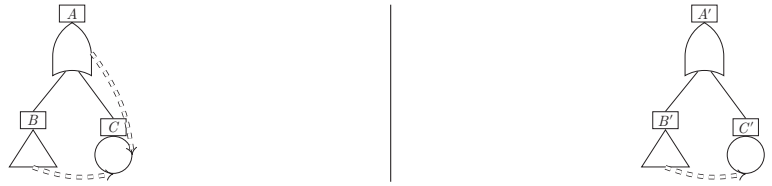
Rewrite rule 25 Superfluous FDEPs from AND.



Input: $\{B_i \mapsto B'_i\}_{i=1}^{2}$
Output: $\{A \mapsto A'\}$

Two triggers always fail simultaneously. The next rule exemplifies the elimination of an FDEP in case multiple FDEPs have shared dependent events. In the DFT on the right-hand side, there are two reasons for OR-gate A to fail (and to trigger C): either C fails first (causing A to fail), leaving nothing left to trigger, or (b) B fails first (triggering C) and A fails. In both cases, the triggering of C via A is superfluous.

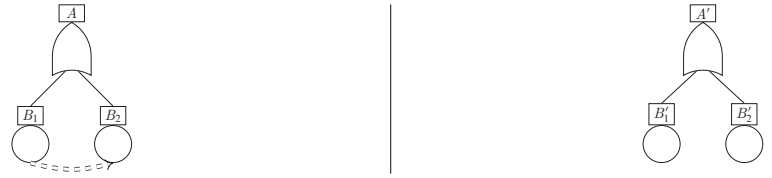
Rewrite rule 26 Superfluous FDEPs from OR.



Input: $\{B \mapsto B', C \mapsto C'\}$
Output: $\{A \mapsto A'\}$

Eliminating FDEPs with common predecessors. Some functional dependencies are superfluous if they have common predecessors. We give two examples focussing on a single common predecessor. The generalisation to multiple common predecessors is not directly supported by these rules, but can be mimicked as illustrated in Fig. 26. The first rule considers an FDEP between two successors of an OR-gate (see below). Here, A fails whenever B_1 fails first, or when B_2 fails. In both cases, the functional dependency between B_1 and B_2 is superfluous. This only works if B_2 has no other predecessors, as otherwise the FDEP might affect the failure of these predecessors. At first sight, it seems that this rule can be simulated using Rules 3 and 24. However, the restrictions from Rule 24 are strong and might prevent the rules from being applicable.

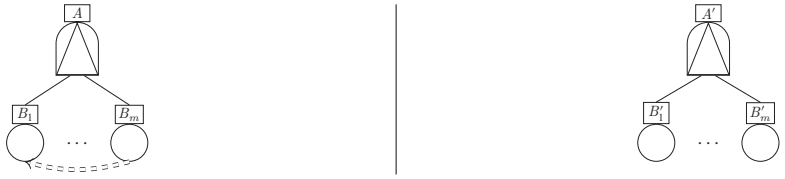
Rewrite rule 27 Removing FDEP between successors of an OR-gate.



Input: $\{B_i \mapsto B'_i\}_{i=1}^{2}$
Output: $\{A \mapsto A'\}$
Context Restriction: NoOtherPreds(B_2)

The next elimination rule for functional dependencies considers an m-ary PAND. If the trigger B_m fails, then either the dependent event failed before or the PAND is rendered infallible, regardless of the subsequent failure of the dependent event.

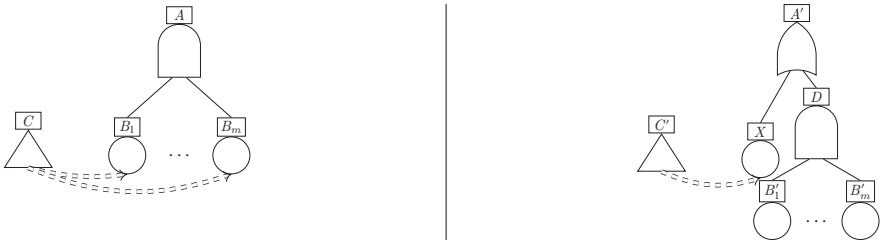
Rewrite rule 28 Eliminating FDEP between successors of a PAND.



Input: $\{B_i \mapsto B_i'\}_{i=1}^m$
Output: $\{A \mapsto A'\}$
Context Restriction: NoOtherPreds(B_1)

Combining FDEP**s.** Sometimes, we can redirect the functional dependencies by the introduction of an additional element event. These events can often be eliminated later. The last rule eliminates functional dependencies in the context of an AND-gate. Consider the rule below. If trigger C occurs, then all dependent events B_1 through B_m fail, causing a failure of AND-gate A. The rhs of the rule mimics this by introducing event X that solely depends on C'. The OR-gate A' fails if either all B_i's fail, or trigger C' occurs. This correctness of this transformation depends on the fact that the successors B_1 through B_m do not have other predecessors than AND-gate A. Further rule applications might eliminate both the FDEP and X.

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: $\{\text{NoOtherPreds}(B_i)\}_{i=1}^m$

This completes the set of rewrite rules. Our set of rewrite rules does not include any rules concerning SPARE gates. This is mainly for the sake of simplicity. Due to activation propagation and claiming, such rules would be either not generally applicable or rather involved. The typical examples of spare-gates do not allow for much simplification, thereby any motivation to provide such rules is limited.

5.4. Correctness

For proving correctness, we introduce the auxiliary concept of *failed under an oracle*. Intuitively, we have no or only partial knowledge of the context of the DFT on which a rule should be applied. As we want equal behaviour for all states, we use an oracle to obtain the state of the input nodes of our rewrite rule. Based on the outcome of the oracle, the failure is propagated through the subDFTs. This propagation is as before.

Definition 5.8 Let $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \to R, \mathfrak{C})$ be a rewrite rule with event elements B and let $B' \supseteq B$. We call a function $f, f\colon B'^{\triangleright} \to \mathcal{P}(V_\mathrm{i})$ such that $f(\pi) \subseteq f(\pi \cdot x)$ for any $x \in B'$, $x \notin \pi$ an *(output-independent input-)oracle for* $\mathfrak{r}$.

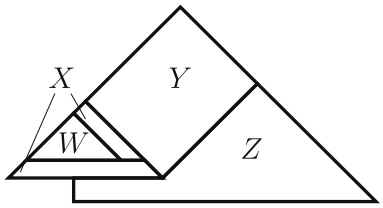


Fig. 27. Illustration for the proof of Theorem 5.1

Definition 5.9 Let F be a DFT, $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ a rewrite rule with event elements B and let $B' \supseteq B$. Let κ be a match homomorphism from $\mathfrak{r}$ to F 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 $\text{Failed}_F[f](\pi)$ of *failed elements considering* f. For $v \in \kappa(V_\mathfrak{i})$, we have that $v \in \text{Failed}_F[f](\pi)$ iff $v \in f(\pi)$. For $v \in V_F \backslash \kappa(V_\mathfrak{i})$, we have that $v \in \text{Failed}_F[f](\pi)$ iff $v \models_F \pi$ given that $v' \models_F \pi$ for all $v' \in f(\pi)$[1].

We define $\text{DepEvents}(\pi)[f]$ as $\text{DepEvents}(\pi)$ where the oracle is taken into account.

5.4.1. Proof obligations without FDEPs

We start with a couple of contexts for rules which neither contain FDEPs nor SPAREs. Notice that this does not mean that these mechanisms are not present in the remainder of the DFT. We only exclude the elements (FDEPs, SPARE-gates) from being present in the applied rule.

Unrestricted context. The easiest context restriction is the unrestricted case. Notice that as we have no FDEPs, we do not have to observe interleavings in which events are triggered. We can focus on equivalent behaviour based on a set of event elements. Notice that a DFT F and the result of rewriting F' range over the same event elements. Moreover, claiming is unaffected by these rules, as we do not allow that SPAREs are matched. It remains to show that the failure behaviour of the *top* remains untouched and that the activation scheme remains untouched.

Theorem 5.1 (Unrestricted context proof obligation) *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{FDEP}} = L_{\mathsf{SPARE}} = R_{\mathsf{FDEP}} = R_{\mathsf{SPARE}} = \emptyset$. *Then* $\mathfrak{r}$ *is valid if*

$$\forall\ \pi \in B'^{\triangleright}.\quad \forall\ f \text{ oracle for } \mathfrak{r}.\ h_r(\text{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \text{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})$$

and for all $\{v, v'\} \subseteq V_\mathfrak{i} \cup V_\mathfrak{o}$ *it holds* v *and* v' *are in the same module iff* $h_r(v)$ *and* $h_r(v')$ *are in the same module.*

Proof sketch. The key steps partition the DFT as in Fig. 27, where $W \cup X$ is the part of the DFT which will be replaced, with X as interface. It is shown that one of the oracles under consideration matches the behaviour of the original DFT. Furthermore, the behaviour of Z is unaffected by any rule which adheres to the requirements, and the behaviour of elements in Y is only affected by nodes in the output interface and by nodes in Z. The activation is guaranteed to be as before for all EEs and SPAREs, using the fact that W does not contain EEs or SPAREs, and all modules are preserved up to elements in W. Now as the DFT ranges over the same EEs and the behaviour is the same, the IMCs induced by the original and the rewritten DFT are isomorphic. The full proof can be found in [Jun15].

Rules with an injective h_r homomorphism and without context restriction are always *symmetric*. This can be easily deduced from the symmetric nature of the proof obligation.

Event dependent failure proof obligation. The oracle may assign any element to fail initially, that is, without the failure of any EE. For an element x in a DFT, this is only possible if a $\mathsf{CONST}(\top)$ element is in the successor closure. The EventDependentFailure context restriction explicitly prohibits the application of rewrite rules if an element x has a $\mathsf{CONST}(\top)$ in the successor closure. We can thus restrict the set of oracles which need to be considered during the proof to those oracles which do not let x fail initially.

[1] We omit a rigorous definition here.

Theorem 5.2 (EventDependentFailure proof obligation) *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{FDEP}} = L_{\mathsf{SPARE}} = R_{\mathsf{FDEP}} = R_{\mathsf{SPARE}} = \emptyset$ *and* $A = \{v \mid \mathit{EventDependentFailure}(v) \subseteq \mathfrak{C}\} \subseteq V_\mathfrak{i}$. *Then* $\mathfrak{r}$ *is valid if*

$$\forall\ \pi \in B'^{\triangleright}.\quad \forall\ f \text{ oracle for } \mathfrak{r} \text{ s.t.}\quad \forall\ a \in A\ a \notin f(\varepsilon).h_r(\text{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \text{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})$$

and for all $\{v, v'\} \subseteq V_\mathfrak{i} \cup V_\mathfrak{o}$ *it holds* v *and* v' *are in the same module iff* $h_r(v)$ *and* $h_r(v')$ *are in the same module.*

The adaptions in the proof theorem are almost trivial. It only has to be shown that the—now restricted—set of oracles suffices to cover any possible scenario (i.e., context). The proof uses the fact that now additional assumptions about the context can be made.

Independent inputs proof obligation. In a similar fashion, the oracle may let any two elements fail simultaneously; in a DFT, two elements can only fail simultaneously if they can fail due to the same EE. The $\mathit{IndependentInputs}$ context restriction excludes such DFTs, thus the set of oracles that need to be considered can be restricted.

Theorem 5.3 (IndependentInputs proof obligation) *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{FDEP}} = L_{\mathsf{SPARE}} = R_{\mathsf{FDEP}} = R_{\mathsf{SPARE}} = \emptyset$ *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*

$$\forall\ \pi \in B'^{\triangleright}.\quad \forall\ f \text{ oracle for } \mathfrak{r} \text{ s.t.}\quad \forall\ a \in A\quad \forall\ x \in B'\ a \not\subseteq f(\pi \cdot x) \backslash f(\pi)$$
$$h_r(\text{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \text{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})$$

and for all $\{v, v'\} \subseteq V_\mathfrak{i} \cup V_\mathfrak{o}$ *it holds* v *and* v' *are in the same module iff* $h_r(v)$ *and* $h_r(v')$ *are in the same module.*

As for the event dependent context, the adaptions in the proof are trivial.

Activation connection proof obligation. Besides restricting the failure-oracle, we often need more liberal requirements on connected interface elements, while still ensuring correct activation. The activation connection restriction allows us to separate two previously connected elements; if their connection—within the given context—is superfluous, we can eliminate it. This allows us to loosen the proof obligation of the second point, where we no longer assure that all connections are kept intact. Notice that this restriction is not tailored toward previously unconnected elements: they have to remain unconnected.

Theorem 5.4 *Let* $\mathfrak{r} = (L \leftarrow (V_\mathfrak{i} \cup V_\mathfrak{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{FDEP}} = L_{\mathsf{SPARE}} = R_{\mathsf{FDEP}} = R_{\mathsf{SPARE}} = \emptyset$ *and* $A = \{\{v_1, v_2\} \mid \mathit{ActivationConnection}(v_1, v_2) \subseteq \mathfrak{C}\} \subseteq \mathcal{P}(V_\mathfrak{i} \cup V_\mathfrak{o})$. *Then* $\mathfrak{r}$ *is valid if*

$$\forall\ \pi \in B'^{\triangleright}.\quad \forall\ f \text{ oracle for } \mathfrak{r}.\ h_r(\text{Failed}_L[f](\pi) \cap V_\mathfrak{o}) = \text{Failed}_R[f](\pi) \cap h_r(V_\mathfrak{o})$$

Furthermore, for all $\{v, v'\} \in \mathcal{P}_2(V_\mathfrak{i} \cup V_\mathfrak{o}) \backslash A$, v *and* v' *are in the same module iff* $h_r(v)$ *and* $h_r(v')$ *are in the same module. For all* $\{v, v'\} \in A$ *it holds that if* v *and* v' *are in the same module then* $h_r(v)$ *and* $h_r(v')$ *are in the same module.*

The proof goes largely as before, only the activation part is adapted. Here, we make a case distinction to see which point of the context restriction applies. When the context ensures that the activation is as before (as the elements are still connected or not in a spare module), the proof is largely analogous to before. If, however, elements which are no longer activated are superfluous, the proof is more involved: it requires quantitative aspects, that is, the proof shows that the two underlying state spaces remain equivalent, as any now disconnected EE is dispensable. It uses ideas similar to those for the proof of Theorem 4.1.

5.4.2. Proof obligation for rules with FDEPs

Whenever FDEPs are modified by a rule, the set of dependent events after some event element trace π may change. We discuss some proof obligation-extensions which can be applied to such rules.

Unrestricted context with unchanged dependent events. If we can assure that the set of dependent events after any EE trace is unchanged, then the original proof of Theorem 5.1 can be re-used with just two technical tweaks. The only dependent events that can possibly change are those where the FDEP is inside the rule. Thus, this again becomes a local condition on the rewrite rule; which formally states that the dependent events—which are necessarily part of V_i—coincide for any oracle.

Theorem 5.5 *Let* $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{SPARE}} = R_{\mathsf{SPARE}} = \emptyset$. *Then* $\mathfrak{r}$ *is valid if*

$$\forall\ \pi \in B'^{\triangleright} \quad \forall\ f \text{ oracle for } \mathfrak{r}$$
$$h_r(\mathrm{Failed}_L[f](\pi) \cap V_\mathrm{o}) = \mathrm{Failed}_R[f](\pi) \cap h_r(V_\mathrm{o}) \ \wedge\ \mathrm{DepEvents}_L[f](\pi) = \mathrm{DepEvents}_R[f](\pi)$$

and for all $\{v, v'\} \subseteq V_\mathrm{i} \cup V_\mathrm{o}$ *it holds* v *and* v' *are in the same module iff* $h_r(v)$ *and* $h_r(v')$ *are in the same module.*

No other predecessors and ineffective FDEP **elimination.** The stronger variant of removal of FDEPs that do not change which events are triggered, is a rule to eliminate those FDEPs that do not propagate their failure outside of the rule. As dependent events are event elements and event elements are in the input-interface of a rule, in general they can have several predecessors which are not matched by the rule. The NoOtherPreds prevents this. For ease of understanding, we restricted ourselves to subDFTs with just one trigger, x, which is also in the (input)-interface (and therefore remains intact). We enforce that no other elements are made triggers by application of the rule.

As x is the only trigger, we only have to consider dependent events after the failure of x. For any interleaving of dependent events in L, the observable failures—i.e., the failures of interface elements which have predecessors outside of the rule—should be simulated by an interleaving of dependent events in R. More formally, that means that if x fails after the occurrence of π, we have a set of dependent events (C) which then fail. For every sequence π' over any subset of these dependent events, the set of interface elements that might have predecessors outside of L and which fail after $\pi \cdot \pi'$ can also fail after some sequence over a subset of dependent events in R.

Theorem 5.6 *Let* $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{SPARE}} = R_{\mathsf{SPARE}} = \emptyset$ *and* $\{\sigma(v)_1 \in V_\mathrm{i} \mid v \in L_{\mathsf{FDEP}}\} = \{x\} \supseteq \{\sigma(v)_1 \in V_R \mid v \in R_{\mathsf{FDEP}}\}$ *for some* x. *Let* $A = \{v \mid \mathit{NoOtherPreds}(v) \subseteq \mathfrak{C}\} \subseteq V_\mathrm{i}\}$. *Then* $\mathfrak{r}$ *is valid if*

$$\forall\ \pi \in B'^{\triangleright} \quad \forall\ f \text{ oracle for } \mathfrak{r}$$
$$h_r(\mathrm{Failed}_L[f](\pi) \cap V_\mathrm{o}) = \mathrm{Failed}_R[f](\pi) \cap h_r(V_\mathrm{o}) \wedge$$
$$x \in \mathrm{Failed}_L[f](\pi) \Rightarrow \quad \forall\ C \subseteq \mathrm{DepEvents}_L[f](\pi) \exists\ C' \subseteq \mathrm{DepEvents}_R[f](\pi)$$
$$\forall\ \pi' \in C^{\triangleright} \exists \pi'' \in C'^{\triangleright}$$
$$h_r(\mathrm{Failed}_L[f](\pi \cdot \pi') \cap V_\mathrm{i} \backslash A \cup V_\mathrm{o}) = \mathrm{Failed}_R[f](\pi \cdot \pi'') \cap h_r(V_\mathrm{i} \backslash A \cup V_\mathrm{o})$$

and for all $\{v, v'\} \subseteq V_\mathrm{i} \cup V_\mathrm{o}$ *it holds* v *and* v' *are in the same module iff* $h_r(v)$ *and* $h_r(v')$ *are in the same module.*

Notice that this elimination affects the underlying state space, thus we have to prove that the underlying state spaces are still equivalent. This proof again uses ideas to argue over the underlying state space also used for Theorem 4.1, as the dependent events are dispensable (given that they are triggered). The proof that the elements are indeed dispensable goes along the lines of the unrestricted context case, in that sense that for any oracle over the input *and any order of triggered events*, the outputs are equivalent.

A proof obligation for rewrite Rule 24. The last proof obligation we discuss here aims at rewrite Rule 24 and variants thereof. First of all, already due to syntactical restrictions, spare modules cannot be merged. Thus, we aim to apply the rule in situations where obviously the activation propagation is untouched. We enforce this by the TopConnected restriction; this allows us to focus on the failure behaviour. The proof obligation is aimed to eliminate FDEPs. In fact, we restrict to ourselves to eliminating FDEPs one by one. We recall that we assume w.l.o.g. only one dependent event per functional dependency. We argued in our discussion of the rewrite rule that we enforce an order independence of the dependent event and that this element has no other predecessors.

Theorem 5.7 *Let* $\mathfrak{r} = (L \leftarrow (V_\mathrm{i} \cup V_\mathrm{o}) \rightarrow R, \mathfrak{C})$ *be a rewrite rule with event elements* B *and let* $B' \supseteq B$. *Let* $L_{\mathsf{SPARE}} = R_{\mathsf{SPARE}} = R_{\mathsf{FDEP}} = \emptyset$ *and* $L_{\mathsf{FDEP}} = \{x\}$ *for some* x. *Let* $\mathit{OrderIndependentNode}(\sigma(x)_2) \subseteq \mathfrak{C}$ *and* $\mathit{NoOtherPreds}(\sigma(x)_2) \subseteq \mathfrak{C}$. *Then* $\mathfrak{r}$ *is valid if*

$$\forall\ \pi \in B'^{\triangleright}.\quad \forall\ f, f' \text{ oracle for } \mathfrak{r} \text{ with } f'(\pi) = f(\pi) \cup \{\sigma(x)_2\}$$
$$\sigma(x)_1 \notin f(\pi) \Rightarrow h_r(\mathrm{Failed}_L[f](\pi) \cap V_\mathrm{o}) = \mathrm{Failed}_R[f](\pi) \cap h_r(V_\mathrm{o}) \wedge$$
$$\sigma(x)_1 \in f(\pi) \Rightarrow h_r(\mathrm{Failed}_L[f'](\pi) \cap V_\mathrm{o}) = \mathrm{Failed}_R[f](\pi) \cap h_r(V_\mathrm{o})$$

and for all $v \in V_\mathrm{i} \cup V_\mathrm{o}$ *it holds that* $\mathit{TopConnected}(v) \subseteq \mathfrak{C}$ *or* $\exists\, v' \in V_\mathrm{i} \cup V_\mathrm{o}$ *with* $\mathit{ModPath}_L(v, v') \neq \emptyset \neq \mathit{ModPath}_R(v, v')$ *and* $\mathit{TopConnected}(v') \subseteq \mathfrak{C}$.

The significant difference to the earlier proof obligation occurs when $\sigma(x)_1 \in \mathrm{Failed}_L[f](\pi)$ and $\sigma(x)_2 \notin \mathrm{Failed}_L[f](\pi)$. These are exactly the cases where the FDEP is usually effective. In those cases, we are only interested in equivalence after the dependent event has been triggered. As $\sigma(x)_2$ is order independent, we can assume that $\sigma(x)_2$ is the first dependent event which fails after π. This failure of $\sigma(x)_2$ should not be noticable from the outside; that is, for any extension of π, the interface elements fail exactly as if $\sigma(x)_2$ would have been triggered.

Combining proof obligations. Notice that the proof obligations are orthogonal, in the sense that the additional restrictions on the activation connection or the oracle do not contradict each other. This makes it trivial to combine several context restrictions.

6. Rewriting DFTs with Groove

Operationalising our rewrite rules is a non-trivial step. Essentially, we need to ensure that the implementation correctly reflects the rules as formally defined. Ideally, one would like to be able to use the rule definitions themselves as executable specifications. In this paper, we have approached that ideal by using graph transformation as a framework in which to encode the rules, as discussed in Sect. 3; see also Fig. 13. To actually execute the rules, we use the tool GROOVE [GdMR+12] as rule engine.

GROOVE uses a graphical syntax for rules in which left and right-hand side graphs are combined into a single graph and colour coding is used to indicate deletions and creations. In particular, dashed blue nodes/edges should be deleted (they are in the left-hand side but not the right-hand side graph), whereas fat green nodes/edges are created upon rule application (they are in the right-hand side but not the left-hand side graph).

However, there is still a conversion required from the graphs as defined in Sect. 3 and the graph rewrite rules generated by translating the DFT rewrite rules in Sect. 4 into the graphs and rules supported by GROOVE. This is necessitated by several factors, such as the fact that many of the DFT and graph rules are given as (infinite) families (indexed by the number of children of a gate), but also because GROOVE works with (strictly) typed graphs, which for instance rule out the use of arbitrary natural numbers as (edge) labels. Therefore, we encode the DFTs as graphs and encode rewrite rules by graph transformation rules in GROOVE. We will briefly review the choices we made in formulating the GROOVE rules, without going into full formal detail.

Typed graphs and child ordering. In GROOVE, graphs and rules are *typed*, meaning that there is a single *type graph* describing the (finite) set of possible node types (comparable to the set of node labels Σ_v) and edge types (comparable to Σ_e, except that an edge type also fixes the source and target node types). Type graphs also support *abstract node types* and *inheritance*, as known from object-oriented modelling and programming.

One particular feature of the DFT graph encoding of Definition 4.2 that is incompatible with typed graphs as supported by GROOVE is the use of natural numbers to encode the ordering of children of a DFT gate.

Figure 28 shows the type graph for GROOVE-encoded DFTs. The italic, dashed node types are abstract. **BE**-typed nodes represent event elements, in which the failure rates and dormancy factors of the attached basic events are given as attributes, whereas **Value**-typed nodes represent CONST($\top$) or CONST($\bot$), depending on the val-attribute (val = true stands for $\top$ whereas val = false stands for $\bot$).

We have used inheritance to good effect by introducing the abstract node type **Gate** with concrete subtypes for the individual gates. This allows us to reflect the sense of rule-families as discussed at the start of Sect. 5.3.

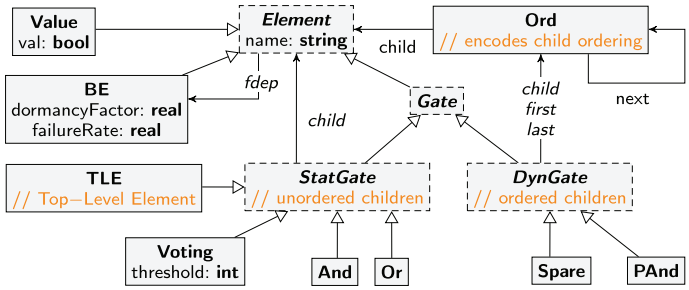


Fig. 28. Type graph for encoded DFTs

Child ordering is encoded in GROOVE by using intermediate auxiliary **Ord**-nodes with next-edges pointing from each child to the next. The children of a gate are reached via child-edges, with auxiliary first- and last-edges to mark the first and last child. For static gates (where the ordering of the children does not matter) we can skip the intermediate **Ord**-nodes, with the automatic effect that the commutativity and associativity of AND, OR and VOT(k) are directly encoded in the graphs.

Finally, FDEP-gates are in GROOVE not represented as nodes at all; instead, we use fdep-edges from the trigger element to all dependent event elements.

As an example, Fig. 29a shows the encoding of a fragment of the Railway Crossing DFT of Fig. 7.

Context restrictions and application conditions. GROOVE rules can include application conditions, in the form of (further) graph structure that has to exist in order for the rule to be applicable. Such application conditions are as expressive as first order logic with transitive closure, and hence are perfectly capable of encoding the context restrictions of the DFT rules. Thus, in the GROOVE encoding, the context conditions are integrated into the rules themselves.

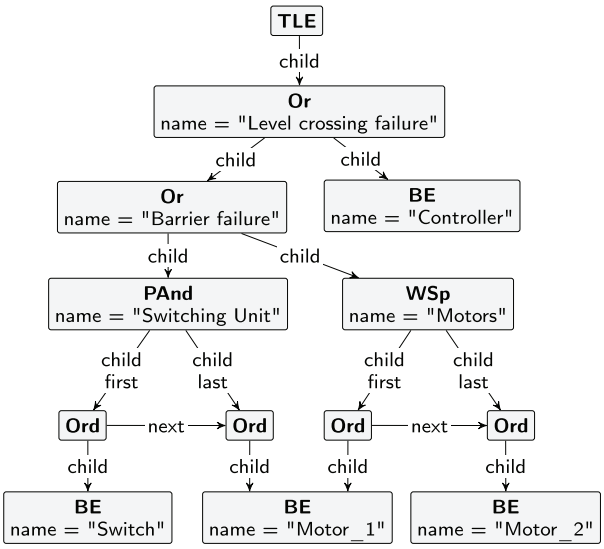
Rule families. Many of the DFT rewrite rules are actually rule families. The translated graph rewrite rules are likewise families, indexed by the number of children of the output gate. Though it would be possible simply to include as many instances of such a family as one is likely to need in practice (say, up to 10 children for each gate), this is very unappealing for several reasons, having to do with maintainability, readability and performance.

Instead, GROOVE offers so-called *nested rules*, which allow parts of a rule to be (existentially or) universally quantified. When matching a rule, every ∀-quantified subrule is matched *as often as possible*, and when applying the rule, each of the subrule images is rewritten simultaneously. This mechanism is very suitable to encode rule families. Moreover, this does away with the additional restriction on DFT matches imposed in Definition 4.6, viz. that all successors of a graph vertex matched by an output element have to be in the matched graph themselves: this is automatically guaranteed by the nature of matching for ∀-quantified subrules.

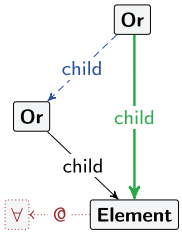
As an example, Fig. 29b shows the GROOVE rule corresponding to DFT Rule 5 (left-flattening). The **Element**-node is matched against all child-edges of lower **Or**, and for each such child a new child-edge is created from the upper **Or**.

Non-structural rules. For the implementation in GROOVE, removing event elements which are not the *top* element (**TLE**) and do not have predecessors is a trivial step. Merging two EEs is slightly more involved as it involves some arithmetic operations on the failure rates; however, GROOVE supports arithmetic operations on numerical attributes, which allows the creation of an **BE** with the required failure rate.

Controlled rewriting. Typically, for a given DFT many different rules are applicable, some of which may be conflicting (in the general sense of rewriting theory [DJ91], meaning that the application of one rule makes the other inapplicable). In other words, as given, our rule system is not confluent: the choice of the next rule to be applied makes a difference in the set of DFTs reachable afterwards.



(a) Fragment of the Railway Crossing DFT (Fig. 7)



(b) OR-left-flattening (Rule 5)

Fig. 29. Example DFT and rewrite rule encodings

Our overall aim of rewriting DFTs can in fact be seen as a search problem in the space of DFTs, where the search steps are rule applications and we search for a reachable DFT that is optimal in the sense of being structurally simple. Because the search space is far too large to lend itself to exhaustive exploration, a well-known tactic is to impose a heuristic that restricts the paths that are actually explored. In our case, we have devised a heuristic that essentially partitions the set of rules into three subsets, which are then applied with descending priority:

1. Garbage-collecting rules (e.g., removal of disconnected elements);
2. Elimination rules (e.g., the left-flattening Rule 5);
3. Structural rewriting (e.g., Rule 20).

More precisely, given a DFT representing the current state of rewriting, we always non-deterministically try to apply one of the rules from group 1; if none are applicable, we try one of the rules from group 2; if none of those are applicable either, we go to the rules from group 3. If, by applying a lower-priority rule, one of the higher-priority ones becomes applicable afterwards, the same principle applies again.

In GROOVE, a heuristic such as the one above can be imposed through so-called *controlled rewriting*. By imposing a control program on a set of rules, one restricts the rules actually considered in any given state. Global rule priorities as described above constitute a fairly simple notion of control; much more sophisticated heuristics can be formulated using sequencing, local priorities, loops and recursion.

7. Experiments

Implementation. We have developed prototypical tool-support[2] exploiting the tools GROOVE [GdMR+12] for graph rewriting, and DFTCalc [ABvdB+13] for the analysis of DFTs. As shown in Fig. 30, our tool chain takes as input a DFT and a measure, i.e., the reliability up to time t, or the MTTF. We translate the DFT into the input format of GROOVE and the output of GROOVE, i.e., the rewritten graph, back into a DFT that then is analysed by DFTCalc. DFTCalc translates the elements of the DFT into processes, and then exploits CADP [GLMS13] for compositional state space generation and reduction (using bisimulation minimisation) of the underlying IMC of the DFT. Finally, the resulting Markov chain is analysed for the user-specified measure by the probabilistic model checker MRMC [KZH+11].

[2] Available online at http://moves.rwth-aachen.de/ft-diet/.

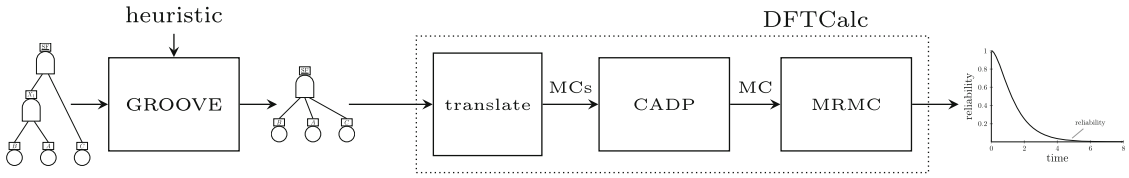


Fig. 30. Tool chain for rewriting and model checking dynamic fault trees

Benchmarks. We have selected a set of benchmarks for DFTs from the literature and from industrial case studies. We have considered four sets of benchmarks that are scalable in a natural manner, to show the scalability of the approach. Several variations of these four benchmarks have been considered, yielding in total 163 cases. We have considered another three industrial cases, yielding an additional 20 cases. All benchmarks are shortly described below; Fig. 31d shows, in brackets after the acronym, for each case how many instances were analysed. Full details and DFTs of the case studies, as well as all statistics can be found at http://moves.rwth-aachen.de/ft-diet/.

For each benchmark, we compared the performance of *base* and *rewriting* (rw) executions, the difference being whether or not the GROOVE component of Fig. 30 was invoked before DFTCalc was run. We investigated the influence of rewriting on (1) the number of nodes in the DFT, (2) the peak memory consumption, (3) the total analysis time (including model generation, rewriting, and analysis), as well as (4) the size of the resulting Markov chain. As can be seen in Fig. 31a–c, rewriting DFTs improves the performance for all these criteria in almost all cases. In particular, 49 cases could be analysed that yielded a time-out or out-of-memory in the base setting.

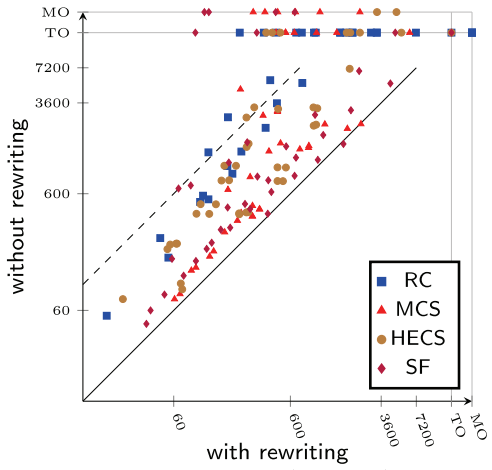
HECS. The *Hypothetical example computer system (HECS)* stems from the NASA handbook on fault trees [SVD+02]. It features a computer system consisting of a processing unit (PU), a memory unit (MU) and an operator interface consisting of hardware and software. These subsystems are connected via a 2-redundant bus. The PU consists of two processors and an additional spare processor which can replace either of the two processors, and requires one working processor. The MU contains 5 memory slots, with the first three slots connected via a memory interface (MI) and the last three connected via another MI. Memory slots either fail by themselves, or if all connected interfaces have failed. The MU requires three working memory slots. We consider a system which consists of multiple (m) (identical) computer systems of which $k \leq m$ are required to be operational in order for the system to be operational. Furthermore, we vary the MI configuration, and consider variants in which all computers have a power supply which is functionally dependent on the power grid.

MCS. The *Multiprocessor computing system (MCS)* contains computing modules (CMs) consisting of a processor, a MU and two disks. A CM fails if either the processor, the MU or both disks fail. All CMs are connected via a bus. An additional MU is connected to each pair of CMs, which can be used by both CMs in case the original memory module fails. The MCS fails, if all CMs fail or the bus fails. The original MCS model was given as a Petri net [MT95], a DFT has been given in [MPBCR06]. The latter includes a power supply (whose failure causes all processors to fail) and assumes the usage of the spare memory component to be exclusive. This is the case we consider. Variations of this model have been given in [ABvdB+13, Rai05]. Based upon these variations we consider several cases. Therefore, we consider a farm of m MCSs of which k are required to be operational. Each MCS contains n CMs (for n uneven, one spare MU is connected to three CMs). Each system has its own power supply. Which is either single power (sp, no redundancy) or double power (dp, one redundant supply for each computer).

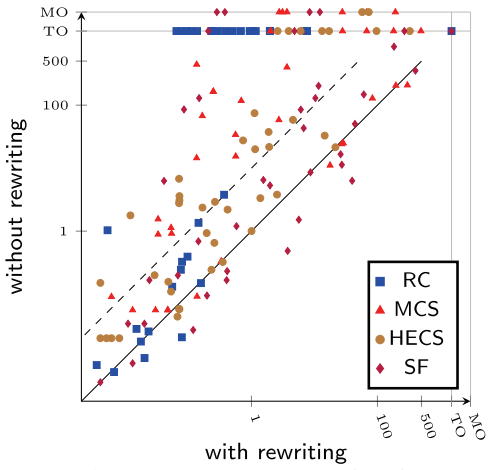
RC. The *Railway crossing (RC)* is an industrial case modelling failures at level crossing [GKS+14] (cf. Fig. 7). We consider an RC that fails whenever any of the sensor-sets fail, or any of the barriers fail, or the controller fails. We obtain scalable versions with b identical barriers and s sets of sensors (each with their own cable which can cause a disconnect). Either the controller failure is represented by a single basic event or by a computer modeled as in HECS.

SF. The *Sensor filter (SF)* benchmark is a DFT that is automatically generated from an AADL (Architecture Analysis & Design Language) system model [BCK+11]. The DFT is obtained by searching for combinations of basic faults which lead to the predefined configurations in the given system. The SF benchmark is a synthetic example which contains a number of sensors that are connected to some filters. The set contains a varying number of sensors and filters.

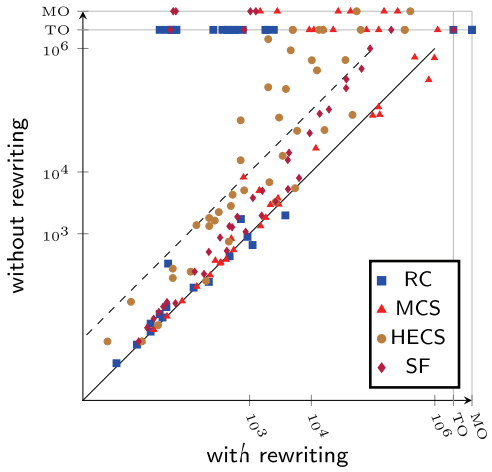
Other case studies. In addition to these scalable benchmarks we have considered other industrial cases such as a *Section of an alkylate plant (SAP)* [CCD+11], a *Hypothetical cardiac assist system (HCAS)* [BD05], and some DFTs (MOV) of railway systems from the Dutch company Movares.



(a) run time (seconds)



(b) memory footprint (MB)



(c) # states in MC

	solved		Σ time (h)			red.
	bs	rw	bs	rw(1)	rw(2)	$\frac{\lvert V_{\text{rw}}\rvert}{\lvert V_{\text{bs}}\rvert}$
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) timing (bs = base)

Fig. 31. Overview of the experimental results on four different benchmark sets

Experimental results. All experiments were run on an Intel i7 860 CPU with 8 GB RAM under Debian GNU/Linux 8.0. Figure 31a indicates the run time for all 163 benchmarks comparing the case with rewriting (x-axis) and without rewriting (y-axis). Note that both dimensions are in log-scale. The dashed line indicates a speed-up of a factor ten. The topmost lines indicate an out-of-memory (MO, 8000 MB) and a time-out (TO, two hours), respectively. Figure 31b indicates the peak memory footprint (in MB) for the benchmarks using a similar plot. The dashed line indicates a reduction in peak memory usage of a factor 10. Finally, Fig. 31c shows the size of the resulting Markov chain (in terms of the number of states), i.e., the model obtained by DFTCalc after bisimulation minimisation. The dashed line indicates a reduction of the Markov chain size by one order of magnitude.

Figure 31d indicates for all seven case studies how many instances could be handled in the base setting (second column), with rewriting (third column), the total time (in hours) in the base (fourth column), the total time with rewriting for those cases that also could be handled in the base (fifth column), and the total time for the cases that could be only dealt with rewriting (sixth column), and the average reduction in number of nodes of the DFTs (last column).

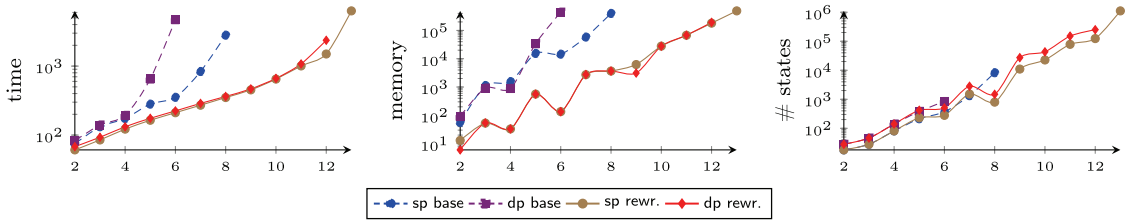


Fig. 32. Effect of rewriting on MCS (n = # CMs, sp/dp = single/double power)

Analysis of the results. In most cases, the reduction of the peak memory footprint as well as the size of the resulting Markov chain is quite substantial, where reductions of up to a factor ten are common with peaks of up to several orders of magnitude. Rewriting enabled to cope with 49 out of 183 cases that yielded a time-out or out-of-memory in the standard setting. For all benchmarks which could be compared, results were verified to be identical up to a precision of $10E{-}8$, which is higher than the precision of $10E{-}6$ guaranteed by DFTCalc, and most results where exactly identical. The 49 cases for which we only obtained a single result were manually verified to be sound. For a few cases, rewriting does not give a memory reduction, see the points below the diagonal in Fig. 31b and c. This is mainly due to the fact that CADP exploits *compositional* bisimulation minimisation, where the order in which sub-Markov chains are composed and minimised is determined heuristically [CHZ08]. It may thus occur that equivalent, but structurally different DFTs yield different minimisation orders (and thus peak memory consumption) and distinct minimal Markov chains. The reduction in state spaces of the final Markov chain are mainly due to the reduction of the number of BEs by using Theorem 4.2. In terms of run time, rewriting comes almost for free. In more than 99% of the cases, rewriting speeds up the model construction and analysis, see Fig. 31a. A more detailed analysis reveals that the graph rewriting with GROOVE is very fast, typically between 7 and 12 s. Most time is devoted to the Markov chain construction and bisimulation minimisation (using CADP). The verification time of the resulting Markov chain with the probabilistic model checker MRMC is negligible. The results summarised in the table in Fig. 31d underline these trends. The scalability of our approach becomes clear in Fig. 32 that shows, for two variants of the MCS benchmark, the time, peak memory usage, and size of the resulting Markov chain (y-axis) versus the number of CMs (x-axis). The left plot shows that analysis time is decreased drastically, whereas the right plot shows that the size of the Markov chain is always very close. Plots for the other case studies show similar improvements. The results indicate that systems with two to four times more components become feasible for analysis.

8. Conclusions and future work

Summary. This paper presented a catalogue of rewrite rules to simplify dynamic fault trees (DFTs) prior to their analysis. The crux of our approach is to exploit graph rewriting as a foundation for DFT rewriting. All rewrite rules preserve the reliability and mean time to failure of the DFT at hand. Experimental application to a large number of benchmarks showed that the significant savings in terms of time and memory consumption can be obtained—up to two orders of magnitude.

Discussion and future work. The rewrite rules preserve two important quantitative measures: the reliability and mean time to failure. Our main focus in developing the rewrite rules was to obtain relatively simple rules whose correctness is not too much involved. These 29 rules together with two simple rules to eliminate and merge basic events gives a rather powerful set of rewrite rules, as substantiated by our experimental results. An alternative would be to exploit the full potential of graph rewriting and use that e.g., DFTs over distinct BasicEvents may also preserve the quantitative measures.

In this paper, we used rewriting prior to the compositional generation of the underlying stochastic process of a DFT, as supported by the software tool DFTCalc [ABvdB+13]. We like to point out that our approach is not tailored to the compositional state space generation. Instead, our rewrite rules can be applied to any state space generation procedure for DFTs, e.g., that supported by Galileo [SDC99], or any competitors thereof. In particular, we firmly believe that rewriting can further improve techniques [PD96, LXL+10, HGH11, Yev11, RGD10] that isolate static sub-trees and is applicable to trees similar to DFTs, e.g., dynamic event/fault trees [Kai05], extended FTs [Buc00] and attack trees [Sch99]. Future work is needed to substantiate these claims, as well as to study completeness of the rewrite rules.

The manual encoding of rewrite rule families into GROOVE is a challenging and error-prone process. Future work allowing to automate this seems a prerequisite in order to support more involved rules.

Finally, we are getting more insight in the calibration of our tool chain. In particular, we like to refine our rewriting heuristics, providing more insight in the which rules to apply in what order to obtain maximal reductions.

Acknowledgements

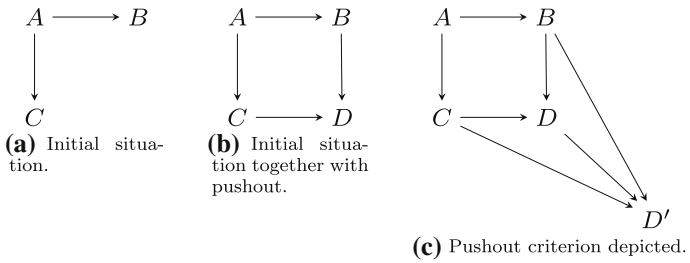
This work has been partially supported by the STW-ProRail partnership program ExploRail under the Project ArRangeer (12238), the NWO Project SamSam, CDZ Project CAP (GZ 1023), and the EU FP7 Grant Agreements No. 318490 (SENSATION) and 318003 (TREsPASS). We acknowledge our cooperation with Movares in the ArRangeer Project.

A. Table of Notation

Symbol	Description	Defined at
$\Sigma^{\triangleright}$	Words over Σ without repetition	p. 11
σ	Successor function	Definition 2.2
$\mathsf{child}(v)$	Successor set of v	p. 10
$\mathsf{parent}(v)$	Predecessor set of v	p. 10
$\mathsf{dec}(v)$	Transient closure of $\mathsf{child}(v)$	p. 11
$\mathsf{anc}(v)$	Transient closure of $\mathsf{parent}(v)$	p. 11
F_K	Elements in F of type K	p. 10
Tp	Defines type of DFT nodes	Definition 2.2
Θ	Attaches failures to event elements	Definition 2.2
top	Top level element (TLE)	Definition 2.2
Mod_r	Module represented by r	p. 12
$\mathrm{ModPath}_F(v, v')$	Path through module from v to v'	Definition 2.5
$\mathrm{Failed}(\pi)$	Failed elements after π	Definition 2.4
$\mathrm{ClaimedBy}(\pi)$	Mapping for current claiming after π	Definition 2.4
$\mathrm{Active}(\pi)$	Elements active after π	p. 12
$\mathrm{Activated}(\pi)$	Active basic elements after π	p. 12
$\mathrm{DepEvents}(v)$	The events that are triggered after v fails	p. 11
$\mathrm{DepEvents}(\pi)$	Triggered not-failed events after π	p. 11
L	Left hand side of a rule	Definition 4.4
R	Left hand side of a rule	Definition 4.4
V_{i}	Input-elements (interface of a rewrite rule)	Definition 4.4
V_{o}	Output-elements (interface of a rewrite rule)	Definition 4.4
h_l	Mapping from interface to lhs of a rule	Definition 4.4
h_r	Mapping from interface to rhs of a rule	Definition 4.4
$\mathfrak{C}$	Set of context-restrictions of a rule	Definition 4.4
$\mathrm{Failed}_F[f](\pi)$	Failed considering input-oracle f	Definition 5.9

B. Graph rewriting definitions

The concept of a *pushout* helps to formalise merging two graphs. The construction is illustrated in Fig. 33. Notation-wise, if $g\colon A \twoheadrightarrow B$ and $h\colon B \twoheadrightarrow C$ are graph morphisms, we use $A \twoheadrightarrow B \twoheadrightarrow C$ to denote the composition $h \circ g$.



(a) Initial situation.

(b) Initial situation together with pushout.

(c) Pushout criterion depicted.

Fig. 33. Pushout construction

Definition B.1 (*Pushout*) Given graphs A, B, C and graph morphisms $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$, a *pushout* consists of a graph and morphisms (D, $B \twoheadrightarrow D$, $C \twoheadrightarrow D$) such that

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

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

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

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

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

The following lemma sketches the construction of a pushout for the graphs we consider here.

Lemma B.1 (Pushout construction) [Ehr79] *Given graphs A, B, C with graph morphisms $b\colon A \twoheadrightarrow B$ and $c\colon A \twoheadrightarrow C$. 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 $\sim_V$.*

Let $D = (V_D, E_D, \mathsf{src}_D, \mathsf{tar}_D, l_D)$. We have that $V_D = V_B \cup V_C / \approx$ and $e \in E_D$ with $\mathsf{src}_D(e) = [s]$, $\mathsf{tar}_D(e) = [t]$ iff $\exists\, v \in [s], v' \in [t]$. $e \in E_B \cup E_C$ with $\mathsf{src}_B(e) = v$ and $\mathsf{tar}_B(e) = v'$ or $\mathsf{src}_C(e) = v$ and $\mathsf{tar}_C(e) = v'$. Moreover lab_D is given by

$$\mathsf{lab}_{v,D}([v]) = \begin{cases} \mathsf{lab}_{v,B}([v]) & v \in V_B \\ \mathsf{lab}_{v,C}([v]) & v \in V_C \end{cases}$$

and, for $e \in E_D$ with $\mathsf{src}_D(e) = [s]$ and $\mathsf{tar}_D(e) = [t]$,

$$\mathsf{lab}_{e,D}(e) = \begin{cases} \mathsf{lab}_{e,B}(e) & \exists\, e' \in E_B \text{ s.t. } \mathsf{src}_B(e') \in [s], \mathsf{tar}_B(e') \in [t], \\ \mathsf{lab}_{e,C}(e) & \exists\, e' \in E_C \text{ s.t. } \mathsf{src}_C(e') \in [t], \mathsf{tar}_C(e') \in [t]. \end{cases}$$

The morphism $f\colon B \twoheadrightarrow D$ is given by $f_V(v) = [v]$ and $f_E(e) = e$ with $\mathsf{src}_B(e) = [s]$ and $\mathsf{tar}_B(e) = [t]$. The morphism $C \twoheadrightarrow D$ is defined analogously.

The application of 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 B.2 (*Rewrite step*) Given two graphs G and H and a rewrite rule $r = (L \twoheadleftarrow K \twoheadrightarrow R)$, 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 B.2 [Ehr79] *Given a rule* $r = (L \leftarrowtail K \twoheadrightarrow R)$ *and a graph* G*, the graph* H *as in Definition* B.2 *can be constructed as follows.*

1. *Find a graph morphism* $\kappa : L \to G$*. Check that the following conditions hold.*
   - *For all* $e \in E_G \backslash \kappa_E(L)$*, it holds that* $\mathsf{src}_G(e), \mathsf{tar}_G(e) \notin \kappa_G(L) \backslash \kappa_G(K)$ (dangling edge condition)*.*
   - *For all* $\{v, v'\} \subseteq V_L$ *and for all* $\{e, e'\} \in E_L, \kappa(v) = \kappa(v') \Rightarrow \{v, v'\} \subseteq V_K$ *and* $\kappa(e) = \kappa(e') \Rightarrow \{e, e'\} \subseteq E_K$ (identification condition)*.*
2. *Get* $D = (V_G \backslash (\kappa_V(L) \backslash \kappa_V(K)), E_G \backslash (\kappa_E(L) \backslash \kappa_E(K)), \mathsf{src}_G \mid E_D, \mathsf{tar}_G \mid E_D, l_D)$ *where* $\mathsf{lab}_D = (\mathsf{lab}_{v,G} \mid V_D, \mathsf{lab}_{e,G} \mid E_D)$*. Get the graph morphism* $K \twoheadrightarrow D$ *as* $\kappa \mid K$*, and the embedding* $D \twoheadrightarrow G$*.*
3. *Construct the pushout KRDH by using Lemma* B.1

References

[ABvdB+13] Arnold F, Belinfante A, van der Berg F, Guck D, Stoelinga MIA (2013) DFTCalc: a tool for efficient fault tree analysis. In: Proc of SAFECOMP, LNCS. Springer, Berlin, pp 293–301.

[BCK+11] Bozzano M, Cimatti A, Katoen J-P, Nguyen VY, Noll T, Roveri M (2011) Safety, dependability and performance analysis of extended AADL models. Comput J 54:754–775

[BCS10] Boudali H, Crouzen P, Stoelinga MIA (2010) A rigorous, compositional, and extensible framework for dynamic fault tree analysis. IEEE Trans Dependable Secur Comput 7(2):128–143

[BD05] Boudali H, Dugan JB (2005) A discrete-time Bayesian network reliability modeling and analysis framework. Reliab Eng Syst Safety 87(3):337–349

[BD06] Boudali H, Dugan JB (2006) A continuous-time Bayesian network reliability modeling and analysis framework. IEEE Trans Reliab 55(1):86–97

[BFGP03] Bobbio A, Franceschinis G, Gaeta R, Portinale L (2003) Parametric fault tree for the dependability analysis of redundant systems and its high-level Petri net semantics. IEEE Trans Softw Eng 29(3):270–287

[BHHK03] Baier C, Haverkort BR, Hermanns H, Katoen J-P (2003) Model-checking algorithms for continuous-time Markov chains. IEEE Trans Softw Eng 29(6):524–541

[BPMC01] Bobbio A, Portinale L, Minichino M, Ciancamerla E (2001) Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. Reliab Eng Syst Safety 71(3):249–260

[Buc00] Buchacker K (2000) Modeling with extended fault trees. In: Proc of HASE, pp 238–246

[CCD+11] Chiacchio F, Compagno L, D'Urso D, Manno G, Trapani N (2011) Dynamic fault trees resolution: a conscious trade-off between analytical and simulative approaches. Reliab Eng Syst Safety 96(11):1515–1526

[CCR08] Contini S, Cojazzi GGM, Renda G (2008) On the use of non-coherent fault trees in safety and security studies. In: Proc European safety and reliability conf (ESREL), pp 1886–1895

[CHZ08] Crouzen P, Hermanns H, Zhang L (2008) On the minimisation of acyclic models. In: CONCUR, vol 5201 of LNCS. Springer, Berlin, pp 295–309

[CSD00] Coppit D, Sullivan KJ, Dugan JB (2000) Formal semantics of models for computational engineering: a case study on dynamic fault trees. In: Proc of ISSRE, pp 270–282

[DBB92] Dugan JB, Bavuso SJ, Boyd MA (1992) Dynamic fault-tree models for fault-tolerant computer systems. IEEE Trans Reliab 41(3):363–377

[DJ91] Dershowitz N, Jouannaud J-P (1991) Rewrite systems. In: van Leeuwen J (ed) Handbook of theoretical computer science. MIT Press, Cambridge, pp 243–320

[DVG97] Dugan JB, Venkataraman B, Gulati R (1997) DIFtree: a software package for the analysis of dynamic fault tree models. In: Proc of RAMS, IEEE, pp 64–70

[EEPT06] Ehrig H, Ehrig K, Prange U, Taentzer G (2006) Fundamentals of algebraic graph transformation. Monographs in Th. Comp. Science. Springer, Berlin

[Ehr79] Ehrig H (1979) Introduction to the algebraic theory of graph grammars (a survey). In: Ng EW, Ehrig H, Rozenberg G (eds) Graph-grammars and their application to computer science and biology, vol 73 of LNCS. Springer, Berlin, pp 1–69

[EPS73] Ehrig H, Pfender M, Schneider HJ (1973) Graph-grammars: an algebraic approach. In: 14th annual symposium on switching and automata theory, IEEE Computer Society, pp 167–180

[GdMR+12] Ghamarian AH, de Mol M, Rensink A, Zambon E, Zimakova M (2012) Modelling and analysis using GROOVE. STTT 14(1):15–40

[GHH+14] Guck D, Hatefi H, Hermanns H, Katoen J-P, Timmer M (2014) Analysis of timed and long-run objectives for Markov automata. Logical Methods Comput Sci 10(3:17):1–29 (2014)

[GKS+14] Guck D, Katoen J-P, Stoelinga MIA, Luiten T, Romijn JMT. (2014) Smart railroad maintenance engineering with stochastic model checking. In: Proc of RAILWAYS. Saxe-Coburg Publications

[GLMS13] Garavel H, Lang F, Mateescu R, Serwe W (2013) CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT 15(2):89–107

[Hec06] Heckel R (2006) Graph transformation in a nutshell. Electr Notes Theor Comput Sci 148(1):187–198

[Her02] Hermanns H (2002) Interactive Markov chains: the quest for quantified quality, vol 2428 of LNCS. Springer, Berlin

[HGH11] Han W, Guo W, Hou Z (2011) Research on the method of dynamic fault tree analysis. In: Proc of ICRMS, pp 950–953

[IEC07] IEC 61025 International Standard: Fault Tree Analysis. 2nd edn, 2006-12, Reference number IEC 61025:2006(E). International Electrotechnical Commission, Geneva, Switzerland

[JGK+15] Junges S, Guck D, Katoen J-P, Rensink A, Stoelinga M (2015) Fault trees on a diet—automated reduction by graph rewriting. In: Proc of SETTA, vol 9409 of LNCS. Springer, Berlin, pp 3–18
[JGKS16] Junges S, Guck D, Katoen J-P, Stoelinga M (2016) Uncovering dynamic fault trees. In: Proc of DSN, IEEE
[Jun15] Junges S (2015) Simplifying dynamic fault trees by graph rewriting. Master Thesis, RWTH Aachen University.
[Kai05] Kaiser B (2005) Extending the expressive power of fault trees. In: Proc of RAMS, IEEE, January, pp 468–474
[KZH+11] Katoen J-P, Zapreev IS, Hahn EM, Hermanns H, Jansen DN (2011) The ins and outs of the probabilistic model checker MRMC. Perform Eval 68(2):90–104
[LXL+10] Liu D, Xiong L, Li Z, Wang P, Zhang H (2010) The simplification of cut sequence set analysis for dynamic systems. In: Proc of ICCAE, vol 3, pp 140–144
[MPBCR06] Montani S, Portinale L, Bobbio A, Codetta-Raiteri, D (2006) Automatically translating dynamic fault trees into dynamic Bayesian networks by means of a software tool. In: Proc of ARES, pp 6
[MR07] Merle G, Roussel J-M (2007) Algebraic modelling of fault trees with priority AND gates. In: Proc of DCDS, pp 175–180
[MRL10] Merle G, Roussel J-M, Lesage J-J (2010) Improving the efficiency of dynamic fault tree analysis by considering gate FDEP as static. In: Proc European safety and reliability conf. (ESREL), pp 845–851
[MRLB10] Merle G, Roussel J-M, Lesage J-J, Bobbio A (2010) Probabilistic algebraic analysis of fault trees with priority dynamic gates and repeated events. IEEE Trans Reliab 59(1):250–261
[MT95] Malhotra M, Trivedi KS (1995) Dependability modeling using Petri-nets. IEEE Trans Reliab 44(3):428–440
[Neu94] Neuts MF (1994) Matrix-geometric solutions in stochastic models—an algorithmic approach. Dover Publications, Mineola
[PD96] Pullum LL, Dugan JB (1996) Fault tree models for the analysis of complex computer-based systems. In: Proc of RAMS, IEEE, pp 200–207
[PH08] Pulungan R, Hermanns H (2008) Effective minimization of acyclic phase-type representations. In: ASMTA, vol 5055 of LNCS. Springer, Berlin, pp 128–143
[Rai05] Raiteri DC (2005) The conversion of dynamic fault trees to stochastic Petri nets, as a case of graph transformation. ENTCS 127(2):45–60
[RGD10] Rongxing D, Guochun W, Decun D (2010) A new assessment method for system reliability based on dynamic fault tree. In: Proc of ICICTA, IEEE, pp 219–222
[RS15] Ruijters E, Stoelinga MIA (2015) Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput Sci Rev 15-16:29–62
[Sch99] Schneier B (1999) Attack trees: modeling security threats. Dr. Dobb's J 24(12):21–29
[SDC99] Sullivan KJ, Dugan JB, Coppit D (1999) The Galileo fault tree analysis tool. In: Proc of Int Symp on fault-tolerant computing, pp 232–235
[SVD+02] Stamatelatos M, Vesely W, Dugan JB, Fragola J, Minarick J, Railsback J (2002) Fault tree handbook with aerospace applications. NASA Headquarters
[Yev11] Yevkin O 2011 An improved modular approach for dynamic fault tree analysis. In: Proc of RAMS, pp 1–5

Received 14 April 2016
Accepted in revised form 22 November 2016 by Cliff Jones, Xuandong Li, and Zhiming Liu