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, Germany
[2] Formal Methods and Tools, University of Twente, 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 by, e.g., probabilistic model checkers. 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.

1 Introduction

Probabilistic safety assessment is common practice in the design and monitoring of safety-critical systems, and often required by law. Typical measures of interest are the system reliability (what is the probability that the system is operational up to time t?) and availability (what is the expected up time?).

Fault tree analysis [38] is one of the most prominent safety assessment technique. It is standardized by the IEC [21], and deployed by many companies and institutions, like FAA, NASA, ESA, Airbus, Honeywell, etc. Fault trees (FTs) model how failures propagate through the system: FT leaves model component failures and are equipped with continuous probability distributions; FT gates model how component failures lead to system failures. Due to, e.g., redundancy, not every single component failure leads to a system failure.

Dynamic fault trees (DFTs) [13,38] are a well-known extension to standard fault trees that cater for common dependability patterns, such as spare management, functional dependency, and sequencing. Analysis of DFTs relies on extracting an underlying stochastic model, such as Bayesian networks [3,6], continuous-time Markov chains (CTMCs) [14], stochastic Petri nets [32,2], and interactive Markov chains [4]. Stochastic model checking is an efficient technique to analyse these models [25,33]. Since the order in which the DFT components fail matters, these approaches severely suffer from the state space explosion problem.

This paper presents a novel technique to reduce the state space of DFTs prior to their analysis. The key idea is to consider DFTs as (typed) directed graphs and manipulate them by *graph transformation* [15], a powerful technique to rewrite graphs via pattern matching. We present a catalogue of 28 (families of) rules that rewrite a given DFT into a smaller, equivalent DFT, having the same system reliability and availability. Various rewrite rules are context sensitive.

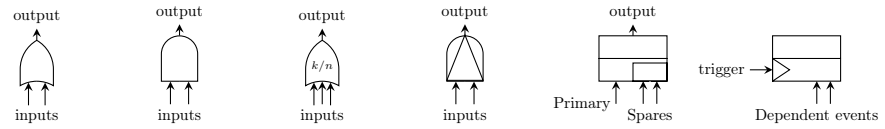
We have implemented our techniques on top of the graph transformation tool GROOVE [17] and the DFT analysis tool DFTCalc [1], yielding a fully automated tool chain for graph-based DFT reduction and analysis. A simple heuristic determines the order to apply the rewrite rules. We have analysed several variations of seven benchmarks, comprised of over 170 fault trees 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 Figures 9(b) and 9(c), page 13). This comes at no run-time penalty: graph rewriting is very fast and the stochastic model generation is significantly accelerated due to the DFT reduction.

Related work. Reduction of fault trees is well-investigated. An important technique is to identify independent static sub-trees [31,26,19,39,35]. These static sub-trees are analysed using efficient techniques (such as BDDs), whereas the dynamic parts require more complex methods, as mentioned above. 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.* [28,29] 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 [32,2] exploit the symmetry in replicated sub-trees while translating DFTs (using graph transformation) to generalized stochastic Petri nets. Finally, DFTCalc exploits compositional aggregation, of (interactive) Markov chains using bisimulation [4].

2 Dynamic fault trees

A dynamic fault tree (DFT) is a tree (or more generally, a directed acyclic graph) that describes how component failures propagate through the system. DFT leaves represent component failures, called basic events (BEs; drawn as ellipses). Fail-safe components and components that have failed already, are denoted respectively by $\mathsf{CONST}(\bot)$ and $\mathsf{CONST}(\top)$.

Gates, depicted in Figure 1, model failure propagation. The static gates OR, AND, $\mathsf{VOT}(k)$ fail if respectively one, all or k of their inputs fail. The PAND, SPARE and FDEP are dynamic gates. A PAND-gate fails if the inputs fail from left to right; if the components fail in any other order, then no failure occurs. A SPARE-gate contains one primary, and one or more spare inputs. If the primary



(a) OR (b) AND (c) VOT(k) (d) PAND (e) SPARE (f) FDEP

Fig. 1. Dynamic fault tree gates.

input fails, then a spare takes over its functionality, putting the spare from dormant into active mode. If all spares have failed as well, then the SPARE-gate fails. Note that (1) primary and spares can be subsystems; and (2) spares can be shared among several components. An FDEP-gate contains a trigger input, which triggers the failure of all its dependent events.

We follow the standard approach and model component failure by exponential probability distributions. Since dormant spare components fail less frequently, we equip each leaf node with a failure rate $\lambda \in \mathbb{R}^+$ and a dormancy factor $\alpha \in [0, 1]$, reducing the failure rate of a dormant component. Thus, the probability for an active component to fail with time t equals $1 - e^{-\lambda t}$; for a dormant component this is $1 - e^{-\alpha\lambda t}$.

Example 1. The DFT in Figure 2 represents the (simplified) failure behaviour of a railway level crossing [18], consisting of three subsystems: the sensors, the barriers and the controller. The crossing fails if either of these subsystems fails. The sensor system fails if at least two out of the four redundant sensors fail Furthermore, there can be a detection problem due to a disconnection of the cables, making all sensors unavailable, modelled by the FDEP-gate No detection specifying that the trigger Disconnection causes the failure of its dependent events Sensor_1–Sensor_4. Finally, the barriers fail if either the main and spare motor fail, modelled by the SPARE-gate Motors, or if the switch and then a motor fails.

Well-formedness. As usual, DFTs must be *well-formed*, meaning: (a) the DFT is acyclic; (b) leaves have leaf-types, other nodes have gate-types; (c) VOT(k)-gates have at least k inputs; (d) the top level event is not an FDEP; (e) FDEPs have no parents; (f) all dependent events of an FDEP are BEs; (g) spare modules, i.e., sub-trees under a SPARE-gate, are independent; (h) primary spare modules, i.e.,

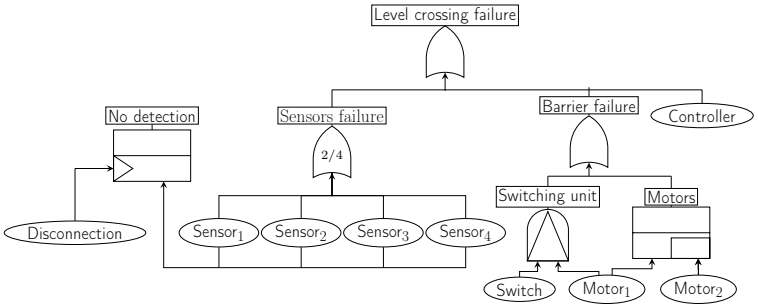
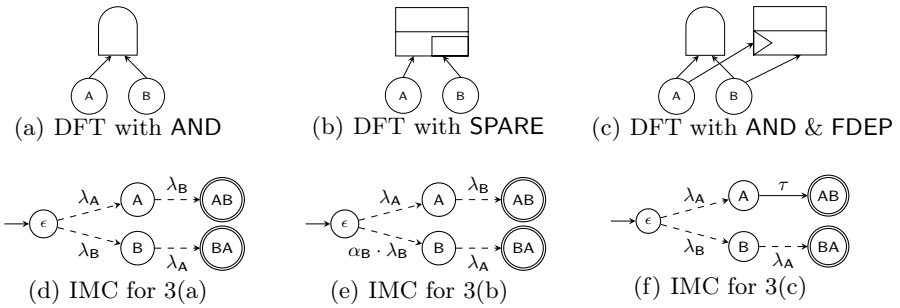


Fig. 2. Railway crossing DFT.



(a) DFT with AND (b) DFT with SPARE (c) DFT with AND & FDEP

(d) IMC for 3(a) (e) IMC for 3(b) (f) IMC for 3(c)

Fig. 3. Example DFT to IMC transformations

sub-trees under the primary input of a SPARE-gate, do not contain CONST($\top$)-nodes; (i) primary spare modules are never shared between SPARE-gates.

DFT semantics. The semantics of a DFT F is — like in [10] — expressed using a transition system $\mathcal{C}_F$, where transitions correspond to the failure of a BE and states to sequences of distinct BEs. As the properties of interest are stochastic in nature, $\mathcal{C}_F$ is an interactive Markov chain (IMC) [20]. This IMC has two types of transitions: interactive (here: immediate τ-transitions) and Markovian (labelled with a rate λ, being the parameter of an exponential distributed delay).

The formal definitions and the construction of $\mathcal{C}_F$ are given in [22]. Here, we illustrate the principle by three small examples, depicted in Figure 3. For ease of reference, we have labelled each state with the sequence of failed BEs. The DFT in Figure 3(a) fails if BE A and then B (upper path in 3(d)) fails, or if first B and then A fails (lower path in 3(d)). The DFT in Figure 3(b) fails if BEs A and B fail, as above. However, if A has not yet failed, then the failure rate of B is reduced by factor α_B. Thus, B is initially dormant, and failes with rate $\lambda_B \cdot \alpha_B$ (see 3(e)). The DFT in Figure 3(c) also fails if BEs A and B fail. However, the failure of A causes B to fail immediately afterwards, as realized by the τ-transition in Figure 3(f).

DFT analysis. A wide variety of qualitative and quantitative DFT analysis techniques are available, see [36] for an overview. We concentrate on two of the most common quantitative measures: the *reliability until a given mission time t,* i.e., the probability that no failure occurs within time t, and the *mean time to failure* (MTTF), i.e., the expected time until the first system failure. Example measures for the DFT in Figure 2 are: "what is the reliability of the level crossing for a time frame of 10 years after deployment?", or "what is the mean time until a first failure of the level crossing occurs?".

We define the probability measures on a DFT F in terms of $\mathcal{C}_F$. Let Fail be the set of states in IMC $\mathcal{C}_F$ in which the top event in F has failed. If a state has multiple dependent events, their order is non-deterministically resolved. This is formalised using a policy (or scheduler) $\mathcal{P}$ on $\mathcal{C}_F$, resulting in a continuous-time Markov chain $\mathcal{C}_F[\mathcal{P}]$.

Definition 1 (Relevant measures). *Given a DFT F, a policy $\mathcal{P}$ on F, and a mission time $t \in \mathbb{R}$, the* reliability of F under $\mathcal{P}$ given t, $\mathit{RELY}^t_{F[\mathcal{P}]}$, *is given*

by $\Pr_{\mathcal{C}_F[\mathcal{P}]}(\Diamond^{\leq t}\mathit{Fail})$. *The* mean time to failure in F under $\mathcal{P}$ $\mathit{MTTF}_{F[\mathcal{P}]}$ *is given by* $\mathsf{ET}_{\mathcal{C}_F[\mathcal{P}]}(\Diamond\mathit{Fail})$, *where* ET *is defined as the expected time.*

3 Rewrite rules for dynamic fault trees

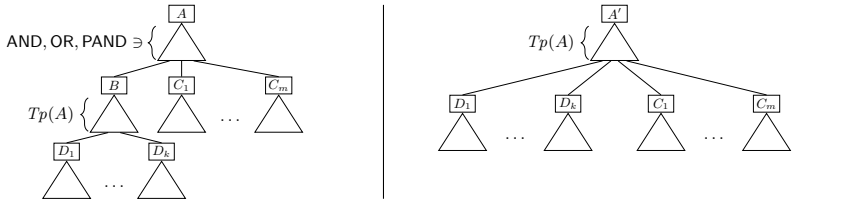
DFTs tend to be verbose. They are often based on the system architecture, therefore reflecting their sub-system structure [38]. Moreover, modern techniques automatically generate DFTs from architectural description languages [7], also yielding rather verbose DFTs. Given that state-of-the-art algorithms construct the underlying IMC by a parallel composition of IMCs corresponding to gates, it is a natural idea to shrink DFTs prior to their analysis. Observe that simplifying *static* fault trees (SFTs) can be done by simple Boolean manipulations [38]. This remains true for FDEPs, but no longer holds in the presence of dynamic gates such as SPARE- and PAND-gates.

We have identified a set of 28 rewrite rules valid for DFTs, which include the (now context-sensitive) rules for SFTs originating from the bounded lattice axioms. Other rules consider combination of dynamic gates which can be simplified. Each rule contains a left-hand (lhs) and a right-hand (rhs) sub-DFT; they are applied by matching the lhs in a given DFT and replacing it by the rhs. Rule application fails if it results in a non-well-formed DFT: in particular, if deletion of a node produces a dangling edge, then the rule cannot be applied. Technically, the mappings between the nodes are defined by a few graph morphisms and context restrictions. For details, see Section 4.

All rules can be used in both directions, i.e., from left to right or from right to left. Below, we present a few key rules and discuss the main issues involved, including the correctness of the rules. In this paper, we display rules only graphically. The complete set of formally defined rules can be found in [22].

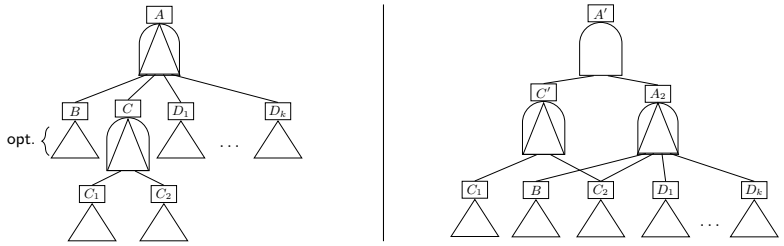
Left-flattening. The first rewrite rule is rather simple and indicates how a DFT with root A (lhs) can be flattened into a DFT with root A' (rhs). In fact, we define a family of rules here, as A can be an AND-, OR-, or PAND-gate. The rule asserts that if A's first successor B has the same type as A, the DFT can be flattened such that B's successors become A's successors (instead of its grandchildren). The rule's correctness in case A is an AND- or OR-gate is obvious; in fact, it can then be applied to any successor of A, not just the first. If A is a PAND-gate, the correctness follows from the fact that the ordering of B's successors is maintained. In this case, the restriction to the first successor is essential.

Rewrite rule 1 Left-flattening of AND-/OR-/PAND-gates



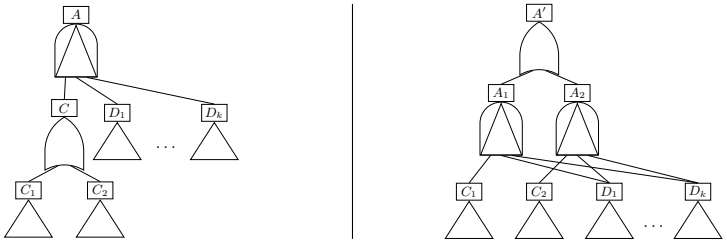
Flattening of PAND-gates. If a PAND-gate has a PAND-successor which is not necessarily its first successor, then the following rule applies. The ordering of C_1 and C_2 is ensured by C' (rhs) whereas the fact that B should fail prior to C_1 and C_2 (in that order) is guaranteed by the PAND-gate A_2 (rhs). This PAND-gate also ensures that B and C_2 should fail before D_1–D_k fail. (Note that successor B is optional.)

Rewrite rule 2 PAND-gate with a PAND-gate as successor



PAND-gates with a first OR-successor. This rewrite rule illustrates a distribution rule for PAND-gates. If the first successor C is an OR-gate with successors C_1 and C_2, then the DFT can be rewritten into a disjunction of two PAND-gates where the first one takes care of the failure of C_1 whereas the second PAND considers the failure of C_2. Although this rule does not give rise to a reduction of the number of DFT elements, it may enable other rewrite rules.

Rewrite rule 3 PAND-gate with an OR-gate as first successor

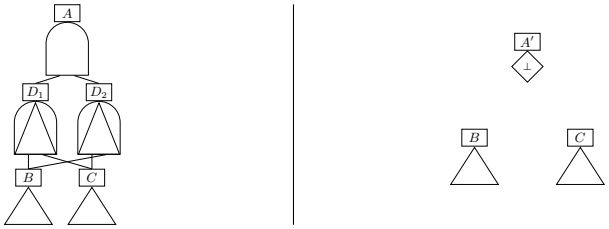


The rules so far are context-free: they can be applied to any sub-DFT matching the lhs. We now give an example of a context-sensitive rewrite rule.

Conflicting PAND-gates. Consider a DFT rooted by an AND-gate which has two PAND-successors whose ordering requirements conflict: D_1 requires B to fail before C, whereas D_2 requires the opposite. In contrast to the rule above, the rewrite rule for this case is context-sensitive: under the assumption that the successors never fail simultaneously (indicated as B and C having independent inputs), the PAND-gates D_1 and D_2 can never both fail. Therefore, the AND-root cannot fail, and the entire DFT is reduced to the DFT on the right. Note that nodes B and C are not eliminated, as they may be connected to other elements of the DFT. However, elimination of the edges may prevent the activation of

elements in the sub-tree. Therefore, the rule may only be applied if, e.g., the sub-trees are activated via another path or if the sub-trees were never activated in the original DFT, as required by the context restriction ActivationConnection.

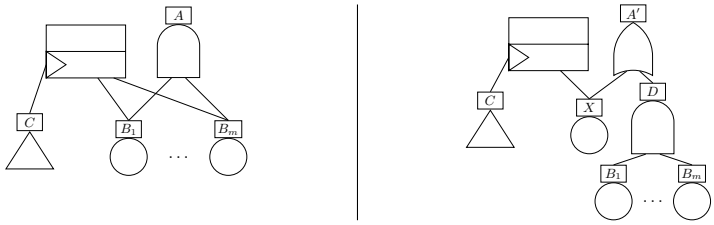
Rewrite rule 4 Conflicting PAND-gates with independent children



Context Restriction: IndependentInputs($\{(B, C)\}$)
ActivationConnection($\{(A, B), (A, C)\}$)

Simplifying of FDEP-gates. If the trigger event C of the FDEP occurs, then the dependent events B_1– B_m all fail, yielding a failure of the AND A. The right DFT emulates this behaviour by adding X – which only fails once trigger C occurs. The root thus fails if either all basic elements B_i fail, or trigger C occurs. This rule is context-sensitive, as its correctness depends on the fact that the successors B_1 through B_m do not have other predecessors besides the AND-gate A. Further rule allow us to get rid of both the FDEP and X in subsequent steps.

Rewrite rule 5 Simplifying FDEP in context of an AND.



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

Rewriting order. A DFT is rewritten by applying a series of rules, with the intention to end up with a simpler, equivalent model. We first consider an example.

Example 2. Consider the DFT in Figure 4 (left). The second DFT results from applying rewrite rule 1 in the reverse direction. Note that this DFT is larger than the first, but enables the application of rule 2, which yields a DFT with a conflicting PAND with independent successors. Applying rule 4 finally allows us to remove the two PAND-gates.

Typically, for a given DFT many different rules are applicable, some of which may be conflicting (in the general sense of rewriting theory [12], meaning that the application of one rule makes the other inapplicable). For instance, in Figure 4,

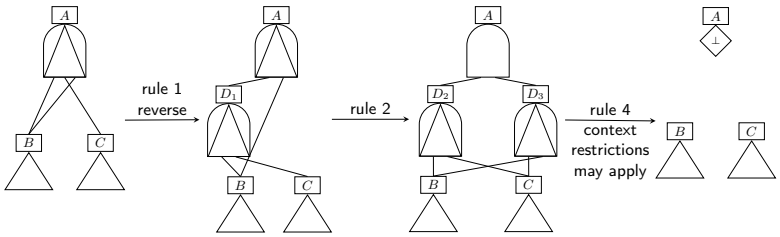


Fig. 4. Steps for rewriting PAND with a duplicate successor.

instead of applying rule 2 as a second step, we could also have applied rule 1, thereby returning to the original DFT. Our overall aim of rewriting any given DFT to a structurally simpler one can in fact be seen as a search problem in the space of DFTs, where the search steps are rule applications. In this paper, we have chosen a fixed, deterministic search strategy. We classified the rewrite rules in three groups, roughly corresponding to the notions of cleaning (e.g., removal of disconnected elements), elimination (e.g., rule 1), and rewriting (e.g., rule 2). Our heuristic is to apply rules from these groups as long as possible, with descending priority.

Correctness. In [22] it is shown that the all 28 rule families are correct, in the sense of preserving the measures-of-interest (see Definition 1).

Theorem 1. *All rewrite rules in [22] preserve all RELY- and MTTF-properties.*

The proof (cf. [22]) amounts to showing that local equivalence of lhs and rhs, consisting of a number of local conditions and context restrictions, implies global equivalence with respect to RELY and MTTF of the source and target DFT of any rule application. These local conditions are then proven to be fulfilled by the rule under consideration — much like the explanations given above.

Theorem 1 means that after rewriting a DFT F into DFT F' using our heuristic strategy (or, indeed, any sequence of rewrite rules whatsoever), we may establish F's properties by analysing the — generally smaller — F'.

4 Rewriting DFTs via graph transformation

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* (GT) as a framework in which to encode the rules, and GROOVE [17] as rule engine. Therefore, we encode the DFTs as graphs and encode rewrite rules by a *sequence* of GT rules (called *recipes* in GROOVE). The operational framework is depicted in Figure 5.

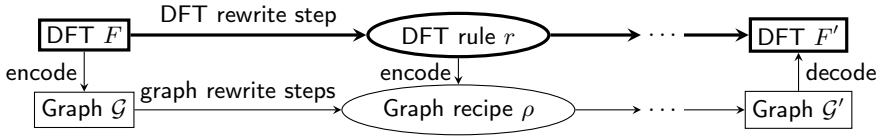


Fig. 5. Operationalising DFT rewriting via graph rewriting.

DFTs as simple graphs. DFTs are directed acyclic node-typed graphs with an ordering imposed on the successors (children) of every node. The graphs in our graph transformation framework, commonly called *simple graphs*, are slightly different: on the one hand they are more basic, since they do not directly support node ordering. Thus, to represent the ordering of dynamic gate successors, we use auxiliary intermediate nodes connected by next-edges. On the other hand, simple graphs are edge-labelled, offering opportunities for compact encodings. In particular, functional dependencies can be encoded as (sets of) edges rather than as nodes, leading from trigger to dependent events.

We have to omit the formal definition of simple graphs and of the DFT encoding. Instead, Figure 6 (slightly simplified for presentation purposes) shows the types of nodes and edges that may occur in a graph-encoded DFT. The italic node types are abstract (in the programming language sense, meaning that only their subtypes can be concretely instantiated). **BE**-typed nodes represent basic events, with corresponding failure rate and dormancy factor, whereas **Value**-typed nodes represent CONST($\top$) or CONST($\bot$), depending on the val flag. **Ord**-nodes encode the child ordering of a dynamic gate, and the fdep-edges the functional dependencies, as discussed above. Figure 7(a) shows the encoding of a fragment of the Railway Crossing DFT of Figure 2.

Rewriting through GT rules. Our GT rules have a structure very similar to the DFT rewrite rules they encode. Each rule consists of a left hand side (lhs) and a right hand side (rhs); the difference between them specifies which nodes or edges should be deleted and which should be created upon application of the rule. GROOVE uses a graphical syntax in which lhs and rhs are combined into a single graph and color coding is used to indicate deletions and creations. In particuar, dashed blue nodes/edges should be deleted, whereas fat green nodes/edges are created upon rule application. For example, Figure 7(b) shows the encoding

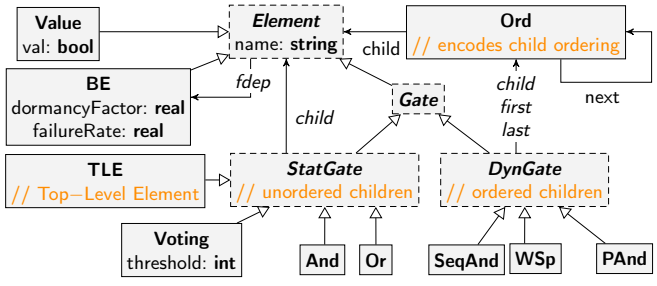
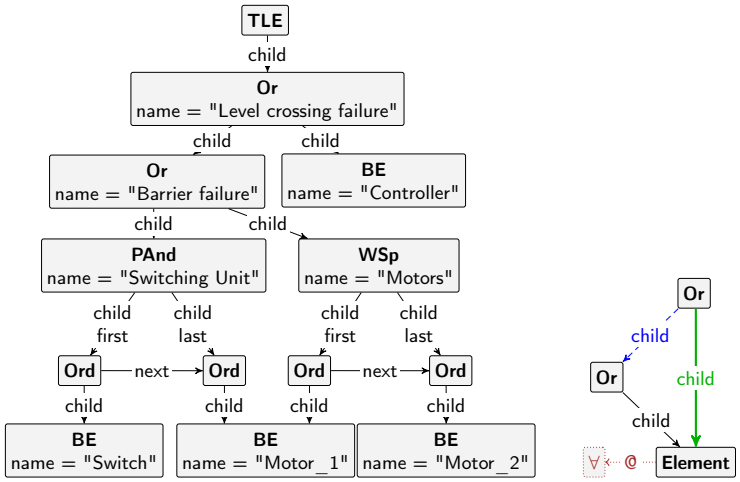


Fig. 6. Type graph for encoded DFTs (slightly simplified)



(a) Fragment of the Railway Crossing DFT (Fig. 2) (b) Flatten (Rule 1)

Fig. 7. Example DFT and rewrite rule encodings.

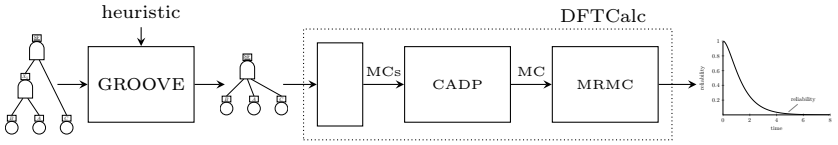


Fig. 8. Tool chain for rewriting and model checking dynamic fault trees.

of the left-flattening of **Or**-gates in rule 1. A feature of GROOVE that is very convenient in this context is the ability to universally quantify over substructures [34]: in the case of left-flattening, *all* successors of the redundant **Or**-gate should be added to the root **Or**. The context-sensitive side conditions of, for instance, rules 4 and 5 are directly encoded as part of the GT rules (which are themselves context sensitive in general). The heuristic search strategy described in Section 3 is implemented in GROOVE using a control program, which is a mechanism to specify a dedicated schedule of rule applications.

Implementation. We have developed prototypical tool-support[1] exploiting the tools GROOVE [17] for graph rewriting, and DFTCalc [1] for the analysis of DFTs. As shown in Figure 8, 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 exploits CADP [16] 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 [24].

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

5 Experiments

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; Table 9(d) 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 Figure 8 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 Figure 9(a)-(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 [38]. 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 [27], a DFT has been given in [30]. 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 [1,32]. 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 [18] (cf. Figure 2). 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.

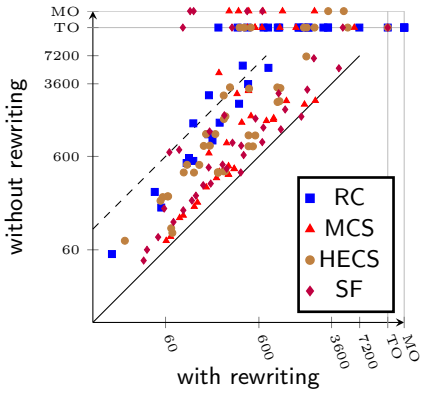
SF. The *Sensor Filter (SF)* benchmark is a DFT that is automatically generated from an AADL (Architecture Analysis & Design Language) system model [7]. The DFT is obtained by searching for combinations of basic faults which lead to the predefined configurations in the given system. The SF benchmark 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)* [9], a *Hypothetical Cardiac Assist System (HCAS)* [5], and some DFTs (MOV) of railway systems from the Dutch company Movares.

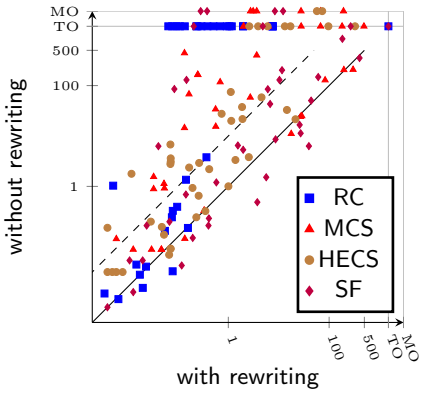
Experimental results. All experiments were run on an Intel i7 860 CPU with 8GB RAM under Debian GNU/Linux 8.0. Figure 9(a) indicates the run time for all 163 benchmarks comparing the case with rewriting (x-axis) and without rewriting (y-axis). Note that both dimensions are in log-scale. The dashed line 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 9(b) 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, Figure 9(c) 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.

Table 9(d) indicates for all 7 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).

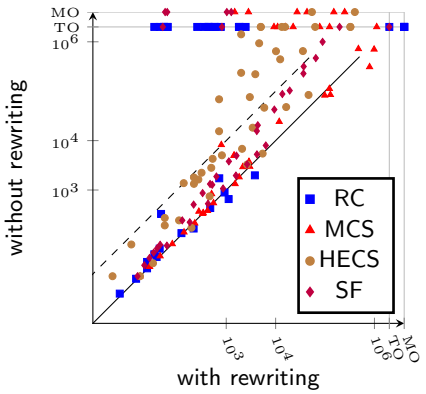
Analysis of the results. In most cases, the reduction of the peak memory footprint as well as the size of the resulting Markov chain is quite substantial, where reductions of up to a factor ten are common with peaks of up to several orders of magnitude. Rewriting enabled to cope with 49 out of 183 cases that yielded a time-out or out-of-memory in the standard setting. For a few cases, rewriting does not give a memory or model reduction, see the points below the diagonal in Figures 9(b) and 9(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 [11]. It



(a) run time (seconds)



(b) memory footprint (MB)



(c) # states in MC

	solved		Σ time (h)			red.
	bs	rw	bs	rw$^{(1)}$	rw$^{(2)}$	$\frac{\lvert V\rvert_{\text{rw}}}{\lvert V\rvert_{\text{bs}}}$
HECS(44)	34	43	11.8	3.3	9.1	1.4
MCS(44)	30	43	9.3	3.7	8.2	1.1
RC(36)	15	31	7.3	5.1	9.3	2.1
SF(39)	31	38	10.1	5.3	7.1	1.5
MOV(8)	3	7	2.3	0.6	0.7	3.4
HCAS(8)	8	8	0.4	0.3	0.3	1.2
SAP(4)	4	4	0.1	0.1	0.1	1.7
total(183)	125	174	41.3	18.4	34.8	1.6

(1) time on instances solved by all.
(2) time on all instances solved.

(d) timing (bs = base)

Fig. 9. Overview of the experimental results on four different benchmark sets.

may thus occur that equivalent, but structurally different DFTs yield different minimisation orders (and thus peak memory consumption) and distinct minimal Markov chains. 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 Figure 9(a). A more detailed analysis reveals that the graph rewriting with GROOVE is very fast, typically between 7 and 12 sec. 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 Table 9(d) underline these trends The scalability of our approach becomes clear in Figure 10 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.

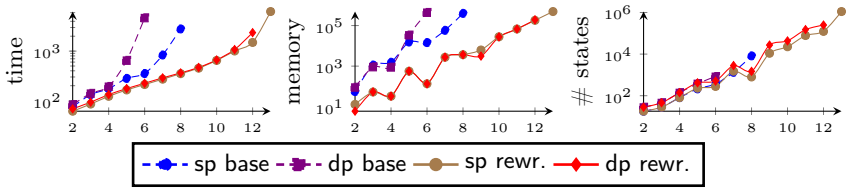


Fig. 10. Effect of rewriting on MCS (n = # CMs, sp/dp = single/double power).

6 Conclusions and future work

This paper presented a novel reduction technique to minimise DFTs using graph rewriting. Application to a large number of benchmarks showed that the savings in terms of time and memory consumption can be drastic, up to two orders of magnitude. Our rewriting approach is applicable too for alternative DFT analysis techniques (rather than DFTCalc). We firmly believe that rewriting can further improve techniques [31,26,19,39,35] that isolate static sub-trees and is applicable to trees similar to DFTs, e.g., dynamic event/fault trees [23], extended FTs [8] and attack trees [37]. Future work is needed to substantiate these claims, as well as to study completeness of the rewrite rules.

Acknowledgement. This work has been supported by the STW-ProRail partnership program ExploRail under the project ArRangeer (12238) and the EU FP7 grant agreements no. 318490 (SENSATION) and 318003 (TREsPASS). We acknowledge our cooperation with Movares in the ArRangeer project.

References

1. F. Arnold, A. Belinfante, F. van der Berg, D. Guck, and M. I. A. Stoelinga. DFTCalc: A tool for efficient fault tree analysis. In *Proc. of SAFECOMP*, LNCS, pages 293–301. Springer, 2013.
2. A. Bobbio, G. Franceschinis, R. Gaeta, and L. Portinale. Parametric fault tree for the dependability analysis of redundant systems and its high-level Petri net semantics. *IEEE Trans. on Softw. Eng.*, 29(3):270–287, 2003.
3. A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla. Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. *Rel. Eng. & Sys. Safety*, 71(3):249–260, 2001.
4. H. Boudali, P. Crouzen, and M. I. A. Stoelinga. A rigorous, compositional, and extensible framework for dynamic fault tree analysis. *IEEE Trans. Dependable Secure Comput.*, 7(2):128–143, 2010.
5. H. Boudali and J. B. Dugan. A discrete-time Bayesian network reliability modeling and analysis framework. *Rel. Eng. & Sys. Safety*, 87(3):337–349, 2005.
6. H. Boudali and J. B. Dugan. A continuous-time Bayesian network reliability modeling and analysis framework. *IEEE Trans. on Reliability*, 55(1):86–97, 2006.
7. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended AADL models. *The Computer Journal*, 54:754–775, 2011.
8. K. Buchacker. Modeling with extended fault trees. In *Proc. of HASE*, pages 238–246, 2000.

9. F. Chiacchio, L. Compagno, D. D'Urso, G. Manno, and N. Trapani. Dynamic fault trees resolution: A conscious trade-off between analytical and simulative approaches. *Rel. Eng. & Sys. Safety*, 96(11):1515–1526, 2011.
10. D. Coppit, K. J. Sullivan, and J. B. Dugan. Formal semantics of models for computational engineering: a case study on dynamic fault trees. In *Proc. of ISSRE*, pages 270–282, 2000.
11. P. Crouzen, H. Hermanns, and L. Zhang. On the minimisation of acyclic models. In *CONCUR*, volume 5201 of *LNCS*, pages 295–309. Springer, 2008.
12. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, pages 243–320. MIT Press, 1991.
13. J. B. Dugan, S. J. Bavuso, and M. A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Trans. Rel.*, pages 363–377, 1992.
14. J. B. Dugan, B. Venkataraman, and R. Gulati. DIFtree: A software package for the analysis of dynamic fault tree models. In *Proc. of RAMS*, pages 64–70. IEEE, 1997.
15. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Th. Comp. Science. Springer, 2006.
16. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107, 2013.
17. A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *STTT*, 14(1):15–40, 2012.
18. D. Guck, J.-P. Katoen, M. I. A. Stoelinga, T. Luiten, and J. M. T. Romijn. Smart railroad maintenance engineering with stochastic model checking. In *Proc. of RAILWAYS*. Saxe-Coburg Publications, 2014.
19. W. Han, W. Guo, and Z. Hou. Research on the method of dynamic fault tree analysis. In *Proc. of ICRMS*, pages 950–953, 2011.
20. H. Hermanns. *Interactive Markov Chains: the Quest for Quantified Quality*. Springer-Verlag Berlin, 2002.
21. Fault tree analysis (FTA). Norm IEC 60050:2006, 2007.
22. S. Junges. Simplifying Dynamic Fault Trees by Graph Rewriting, 2015. Master Thesis, RWTH Aachen University.
23. B. Kaiser. Extending the expressive power of fault trees. In *Proc. of RAMS*, pages 468–474. IEEE, Jan. 2005.
24. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perf. Ev.*, 68(2):90–104, 2011.
25. M. Z. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *SFM*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.
26. D. Liu, L. Xiong, Z. Li, P. Wang, and H. Zhang. The simplification of cut sequence set analysis for dynamic systems. In *Proc. of ICCAE*, volume 3, pages 140–144, 2010.
27. M. Malhotra and K. S. Trivedi. Dependability modeling using Petri-nets. *IEEE Trans. Rel.*, 44(3):428–440, 1995.
28. G. Merle and J.-M. Roussel. Algebraic modelling of fault trees with priority AND gates. In *Proc. of DCDS*, pages 175–180, 2007.
29. G. Merle, J.-M. Roussel, J.-J. Lesage, and A. Bobbio. Probabilistic algebraic analysis of fault trees with priority dynamic gates and repeated events. *IEEE Trans. Rel.*, 59(1):250–261, 2010.
30. S. Montani, L. Portinale, A. Bobbio, and D. Codetta-Raiteri. Automatically translating dynamic fault trees into dynamic Bayesian networks by means of a software tool. In *Proc. of ARES*, pages 6–, 2006.

31. L. L. Pullum and J. B. Dugan. Fault tree models for the analysis of complex computer-based systems. In *Proc. of RAMS*, pages 200–207. IEEE, 1996.
32. D. C. Raiteri. The conversion of dynamic fault trees to stochastic Petri nets, as a case of graph transformation. *ENTCS*, 127(2):45–60, 2005.
33. A. K. I. Remke and M. I. A. Stoelinga, editors. *Stochastic Model Checking*, volume 8453 of *LNCS*. Springer-Verlag, 2014.
34. A. Rensink and J.-H. Kuperus. Repotting the geraniums: on nested graph transformation rules. volume 18 of *ECEASST*. EASST, 2009.
35. D. Rongxing, W. Guochun, and D. Decun. A new assessment method for system reliability based on dynamic fault tree. In *Proc. of ICICTA*, pages 219–222. IEEE, 2010.
36. E. Ruijters and M. I. A. Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, 15-16(0):29–62, 2015.
37. B. Schneier. Attack trees: Modeling security threats. *Dr. Dobb's J.*, 24(12), 1999.
38. M. Stamatelatos, W. Vesely, J. B. Dugan, J. Fragola, J. Minarick, and J. Railsback. *Fault Tree Handbook with Aerospace Applications*. NASA Headquarters, 2002.
39. O. Yevkin. An improved modular approach for dynamic fault tree analysis. In *Proc. of RAMS*, pages 1–5, 2011.