Advancing Dynamic Fault Tree Analysis - Get Succinct State Spaces Fast and Synthesise Failure Rates

Matthias Volk(✉), Sebastian Junges, and Joost-Pieter Katoen

Software Modeling and Verification, RWTH Aachen University, Aachen, Germany
matthias.volk@cs.rwth-aachen.de

Abstract. This paper presents a new state space generation approach for dynamic fault trees (DFTs) together with a technique to synthesise allowed failures rates in DFTs. Our state space generation technique aggressively exploits the DFT structure — detecting symmetries, spurious non-determinism, and don't cares. Benchmarks show a gain of more than two orders of magnitude in terms of state space generation and analysis time. Our approach supports DFTs with symbolic failure rates and is complemented by parameter synthesis. This enables determining the maximal tolerable failure rate of a system component while ensuring that the mean time of failure stays below a threshold.

1 Introduction

Fault tree analysis is a prominent technique in reliability engineering. Dynamic fault trees (DFTs) [1,2] are an expressive model catering for common dependability patterns, such as spare management, functional dependencies, and sequencing. The *state space generation* process is one of the main bottlenecks in DFT analysis. DFT analysis mainly focuses on the mean time to failure — what is the expected time of the failure? — and reliability — how likely is the system operational up to time t? These analyses require DFTs where all component failure rates are known. In practice, this rarely holds. Thus, a relevant question is to *synthesise* the allowed component failure rates ensuring a given mean time.

This paper presents three main advances to state-of-the-art DFT analysis: (1) fast generation of succinct state spaces, (2) the analysis of several measures-of-interest that go beyond mean time and reliability, and (3) the synthesis of (possibly partially) unknown failure rates in DFTs for mean time and more.

Fast Generation of Succinct State Spaces. Our approach is a modern version of one of the first DFT semantics [3] as used in the `Galileo` tool [4] that caters for possible *non-determinism*, as in [5]. In all these approaches, a state space, i.e., a Markov model, is built. This leads to a precise representation of the DFT and allows for off-the-shelf analysis tools. The major drawback is the typically huge state space involved – which has lead to some state-space

A. Skavhaug et al. (Eds.): SAFECOMP 2016, LNCS 9922, pp. 253–265, 2016.
DOI: 10.1007/978-3-319-45477-1_20

free approximation techniques, an overview is given in [6]. To obtain succinct state spaces, we tailor two successful techniques from the field of model checking — symmetry reduction [7] and partial-order reduction [8, Chap. 8] — to DFTs, and combine this with don't care detection. We *aggressively exploit the DFT structure*: detect symmetries, i.e., isomorphic sub-DFTs and stochastic independencies while pruning sub-DFTs that become obsolete (don't care) after the occurrence of some faults. This is combined with *detecting superfluous nondeterminism* such that certain failure orderings are irrelevant yielding a simpler and cheaper analysis.

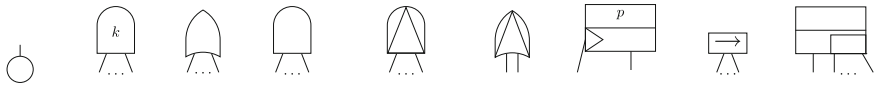
Beyond Reliability and Availability. By exploiting powerful state-of-the-art quantitative model checking techniques [8, Chap. 10] we support a broad range of measures-of-interest. This includes reliability and mean time to failure (MTTF), the probability to reach a certain DFT configuration e.g., where certain subDFTs have failed and others have not, conditional MTTF — what is the MTTF given that certain DFT elements failed? — and the variance of the time to failure.

Failure Rate Synthesis. We support DFTs whose failure rates are (possibly partially) unknown. These unknown (or: symbolic) rates are represented by parameters, or functions thereof; e.g., components may fail with rate λ, 2λ, etc., where λ is unknown. Our slim state space generation techniques support symbolic rates. We complement this by a sound and complete technique to *synthesise* all values of symbolic rates that ensure the MTTF (and various other measures) to be below a given threshold. To the best of our knowledge, this is the first failure rate synthesis technique for DFTs. In addition, the *sensitivity* of the MTTF on the symbolic rates can be determined, as in alternative techniques [9].

Experimentation. We have realised a prototypical implementation of the aforementioned techniques. In addition to the original DFT elements in `Galileo`, we support probabilistic dependencies [10], nested spares [5] and priority or-gates [11]. Experiments have been conducted on all benchmark DFTs from [12]; a rich collection of DFTs gathered from the literature and from industrial case studies. The experiments reveal that our slim state space generation technique significantly outperforms the best competitor for DFTs, the tool `DFTCalc` [13]. For a majority of the benchmarks, our approach yields a speed-up of two to four orders of magnitude. Failure rate synthesis works for the moderately-sized models in the literature (up to 20 basic events) with up to three unknown rates.

2 Dynamic Fault Trees

Fault trees (FTs) are directed acyclic graphs with typed nodes. The leaves, i.e., nodes without successors (or: *children*), are *basic events* (BEs). All other nodes are *gates*. The *top event* (or: root) is a specifically identified node. An FT fails, if its top event fails. For the sake of simplicity, we assume that BEs represent



(a) BE (b) VOT_k (c) OR (d) AND (e) PAND (f) POR (g) PDEP (h) SEQ (i) SPARE

Fig. 1. Node types in ((a)–(d)) static and (all) dynamic fault trees.

component failures. Initially, every BE is *operational*; it *fails* if the event occurs. A gate fails if its *failure condition* over its children is fulfilled. The key gate for static fault trees (SFTs) is the *voting* gate (denoted VOT_k) with *threshold* k. The failure condition for a node x of type VOT_k is given by "x fails, if k of its children have failed". A VOT_1 gate equals an OR-gate, while a VOT_k with k children equals an AND-gate. These gates are shown in Fig. 1(b)–(d).

2.1 Dynamic Nodes

To overcome the limitations [6] of SFTs, several extensions commonly referred to as *Dynamic Fault Trees* (DFTs) have been introduced. A main feature of these extensions is that they feature an internal state, e.g., the order in which events fail influences the internal state, and thus whether the top event has failed. The extensions introduce several new node types; we categorise them as *priority gates*, *dependencies*, *restrictions*, and *spare gates*.

Priority Gates. Priority gates extend static gates by imposing a condition on the ordering of failing children. A *priority-and* (PAND) node fails if all its children have failed in the order from left to right. Figure 2(a) depicts a PAND with children A and B. It fails if A fails first and then (or simultaneously) B fails. If B fails first, the PAND becomes *fail-safe*. The *priority-or* (POR) node [11] only fails if the left-most child fails before any of the other children does. Priority-gates allow for order dependent failure propagation.

Dependencies. Dependencies do not propagate a fault to their parents but are triggered by their first child. Upon triggering, they affect some BEs, the dependent events. We consider *probabilistic dependencies* (PDEPs) [10]. Once the trigger of a PDEP fails, its dependent events fail with probability p. Figure 2(b) shows a PDEP where the failure of trigger A causes a failure of BE B with probability 0.8 (provided it has not failed before). *Functional dependencies* (FDEPs) are PDEPs with probability one.

Restrictions. Restrictions do not propagate failures but rather limit possible failure propagations. *Sequence enforcers* (SEQs) assure that their children only fail from left to right. This differs from priority-gates that do not prevent certain orderings, but only propagate if an ordering is met. The DFT in Fig. 2(c) fails if A and B have failed (in any order) but the SEQ enforces that A fails prior to B. This DFT is never fail-safe.

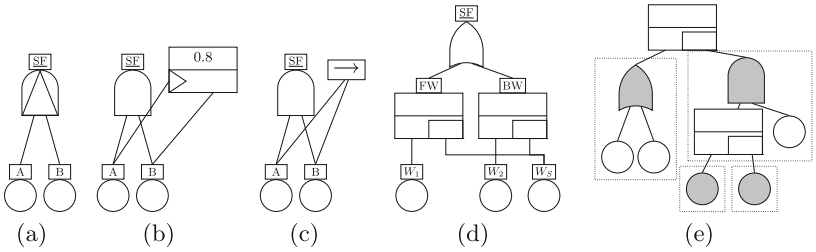


Fig. 2. Simple examples of dynamic nodes.

Spare Gates. Spare-gates (SPAREs) are the most complex gates in DFTs. Consider the DFT in Fig. 2(d) modelling (part of) a motor bike with a spare wheel. If either wheel fails, the motor bike fails. Both wheels can be replaced by the spare wheel but not both. The spare wheel is less likely to fail as long as it isn't used (warm). Assume the front wheel fails. The spare wheel is available and used, and its failure rate is increased (hot). If any other wheel fails, then no spare wheels are available anymore, and the SPARE and the DFT fails.

SPAREs have a child they use. If this child fails, the SPARE tries to use a spare child (left to right) — a process we call *claiming*. Only operational children that are not used by another SPARE can be claimed. If claiming fails, the SPARE fails. This behaviour is extended by an *activation mechanism*. As in [5], SPAREs may have (independent) subDFTs as children. This includes nested SPAREs. A *spare module* is a set of nodes linked to a child of a SPARE via a path without an intermediate SPARE. Every leaf of a spare module is either a BE or a SPARE. Each child of a SPARE thus represents a spare module, cf. Fig. 2(e) where boxes are spare modules and shaded nodes are the representatives. SPAREs which are not nested are *active*. For each active SPARE, all nodes in the spare module of the used child are also active. BEs which are active fail with their active failure rate, BEs which are passive fail with their passive failure rate (warm events) or cannot fail (cold events). More details can be found in [6].

2.2 Syntactic Restrictions

We are rather liberal w.r.t. dynamic gates, but have to impose syntactic restrictions as in [13] to exclude DFTs with undefined behaviour. These restrictions are: (a) VOT_k have at least k children; (b) the top level event is a gate or a BE; PDEPs and restrictions have no parents; (d) all dependent events are BEs; (e) spare modules, i.e., subDFTs under a SPARE, do not overlap; (f) primary spare modules are not shared between SPAREs.

3 State Space Generation

The goal for our state space generation is to produce a Markov model which is subject to further analysis. As operational model, we use *Markov Automata*.

3.1 Markov Automata

Markov Automata (MA) [14] extend continuous-time Markov chains (CTMCs) with non-determinism. MA are state transition systems whose transitions between states are either labeled with rates (i.e., non-negative real numbers), or with actions. The former transitions specify a random delay and correspond to the failures in DFTs; the latter are used to select the handling of a triggered PDEP. Delay transitions relate a source state with a target state; action transitions relate a state to a probability distribution over states. An action transition thus yields a new state with a given likelihood. MA are a slight variant of the operational model for DFTs used in [5]; they differ in allowing discrete probabilistic branching which are used to model PDEPs. We introduce MAs by example.

Figure 3 shows an MA for a coffee machine, used by inhabitants of room A (IoA) and B (IoB). IoA (IoB) arrive at the machine at a rate of 5 IoA/hour (3 IoB/hour). They can either have coffee or espresso. All IoA want espresso (action we), while IoB non-deterministically want coffee (action wc) or espresso. IoB wanting espresso are with probability 0.1 too sleepy and select coffee. Users always get their selected product (ge, gc). In state s_0, either an IoA or an IoB arrives at the machine (evolving into s_1, s_2). In state s_1 espresso is selected, whereas in s_2 a choice between actions we and wc is made. Selecting we in s_2 results in s_3 with probability 0.9 and in s_4 with probability 0.1. The user then gets the product and the automaton returns to initial s_0. For simplicity, the products' preparation time is not modelled.

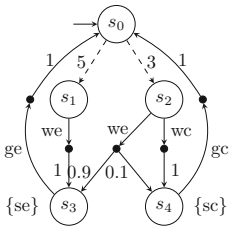


Fig. 3. Example MA.

3.2 State Space Generation

As in Galileo, we construct a *fault tree automaton* (FTAut) from a DFT. We then translate the FTAut to an MA, which we further simplify and analyse. The FTAut consists of states and labelled transitions.

States. We give each node in the DFT a unique id. A state in the FTAut is a mapping from ids to its status: *operational* (OP), *failed* (F), *fail-safe* (FS), or *don't care* (X). Additionally, we store the *currently used child* (CUC) of operational SPAREs and for spare module representatives their *activity*, i.e. whether the module is active (A) or passive (P). We initialise all nodes as operational, the CUCs and activate modules as described in Sect. 2.

Transitions. State changes originate from the failure of BEs. As the probability of two rate-governed BEs to fail simultaneously is zero, BEs never fail simultaneously. When considering dependencies, this assumption no longer has to hold. To avoid problems with causalities as described in [6], and to directly resolve spare

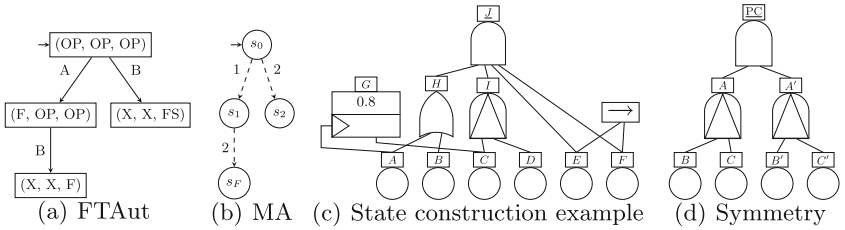


Fig. 4. Dedicated examples.

races [6], we assume that dependent events fail immediately after the triggering BE. W.l.o.g. we assume that PDEPs have a single dependent event.

Given a source state and an operational BE x that fails, we copy the source state and additionally mark x with F, and compute the target state. In a bottom-up fashion, we iterate over the gates. For each gate, we check the failure condition. If the failure condition holds, we mark the gate as failed. If a CUC of a SPARE fails, we iterate over its remaining children and check whether they are not listed as the CUC of any of their parents and whether they are still operational. If so, we update the CUC, otherwise, we mark the SPARE as failed. We iterate over all restrictions, and check whether any of their failure conditions hold; if so, we skip the transition at hand. We then reiterate over all gates, and check if the fail-safe condition holds (i.e. if it cannot fail in the future), we mark the gate FS. We then iterate top-down over all nodes. If all parents of a node are either failed or fail-safe, we mark the node as don't care (DC-propagation).

Example 1. The FTAut of the DFT in Fig. 2(a) is given in Fig. 4(a). Initially, all nodes are operational. If B initially fails, the PAND becomes fail-safe, and thus A and B both become don't care. The resulting state is (X, X, FS). If A however initially fails, B and the PAND remain operational. An additional failure of B then causes the top event to fail. DC-propagation yields the state (X, X, F).

Now consider Fig. 4(c). Initially, every node is operational. A's failure causes H to fail and makes B don't care. This yields a transition from the initial state to state (F, X, OP, OP, OP, OP, OP, F, OP, OP). In this state, the PDEP is triggered, yielding a state (with probability 0.8) in which C failed, and the same state (with probability 0.2) as C does not fail. A failure of D in the initial state does not trigger a failure of the PAND I; in fact I becomes fail-safe, and this is propagated to J, i.e., DC-propagation marks all children (and their children) X. This together yields a transition from the initial state to a state in which all nodes are marked X. Finally, from the initial state, propagating a failure of node F is discarded as the restriction fails (by F failing before E.)

The initial state for nodes $(W_1, W_2, W_S, \mathrm{FW}, \mathrm{BW}, \mathrm{SF})$ in Fig. 2(d) is (OP, OP, OP, W_1, W_2, OP) where W_1, W_2 are the CUCs and as initially the CUCs are active, the activity for W_1, W_2, W_S is given as (A, A, P). A failure of W_1 is propagated to FW. As its CUC fails, it checks further children. W_S is operational and not a CUC, therefore, the resulting state is (F, OP, OP, W_S, W_2, OP) and

(A, A, A). From that state, W_2's failure yields (F, F, OP, W_S, F, F) after failure propagation, as the only remaining child of BW is already claimed. DC-propagation yields the state (F, X, OP, W_S, X, F) and (A, A, A).

As rate-governed transitions have probability 0 to fire at time 0, we either have immediate transitions or rate transitions. Thus, for each state, we check if any PDEPs are triggered. If so, we mark the state as *immediate* and add two outgoing transitions for each triggered PDEP: One where the PDEP transmits the failure and one where it doesn't. Otherwise, we mark the state as *Markovian*, and add transitions for each BE which has (in the given state) a failure rate $\neq 0$.

Translation. The translation from the FTAut to the MA is straightforward, cf. Fig. 4(b). The state spaces of the FTAut and the MA are equal. Each MA state is labeled with the status of the DFT nodes. For Markovian states, each transition labelled with a BE x is translated into a delay transition with the failure rate of x as its rate. For BEs in passive spare modules, we take their passive failure rate. Each immediate state has a non-deterministic choice over triggered PDEPs in the DFT. Each PDEP leads to a probabilistic branching, where with probability p the PDEP propagates the failure, whereas with $1-p$ it does not.

3.3 Optimisations

Technical Aspects. We use a selection of well-known techniques to reduce the overhead of propagation: The states are encoded as bit-vectors, and during exploration, we use an expanded state representation. By exploiting depth-first search, we keep the set of states that we explore later on small. Work lists keep only track of the nodes we need to consider. Overriding failed and fail-safe nodes by don't care, we merge states which differ only in their past behaviour, but not in their future behaviour. Afterwards, the state space is reduced by bisimulation.

Partial-Order Reduction. In many DFTs, the actual order in which subsets of BEs fail is not crucial. We exploit this for dependencies, where — instead of exploring all interleavings over the triggered events — we aim to only explore a single order. We adapt a technique called (static) *partial order reduction* [8] to DFTs. Based on a static analysis, we identify which dependencies can be executed in arbitrary order, and expand only a canonical order.

State Elimination. In MA, we can eliminate probabilistic branching by adopting a *state elimination* technique as used in [15]. In particular, this allows us to reduce MA without non-deterministic branching to CTMCs, which can be analysed much faster as non-determinism is absent.

Modularisation. The use of modularisation in FTs has been proposed in [16]. It identifies independent subtrees in the DFT, analyses them separately, and combines the results to a final result. If applicable, it is extremely powerful.

Symmetry Reduction. Many parts in DFTs are symmetric. This can be exploited (cf. [7]) as follows. Given a successfully detected symmetry, we use the fact that a fault has an analogous effect in symmetric parts. Moreover for isolated symmetric parts, if the node identities are not used in the analysis and the parts are only connected to the remaining DFT via the same node, we *exchange* the states of the parts, and thus assume that a fault in a symmetric part happened in an equivalent DFT. In the DFT in Fig. 4(d), we find two symmetric parts (the subtrees of A and A'), which are independent. If we are only interested in the top level, we can use the exchange technique. That is, if both symmetric parts are in equivalent states (e.g., the initial state) and A' fails, we assume that A failed instead. Now, the two parts are not in an equivalent state. However, after the additional failure of A', the two parts are in an equivalent state again.

4 Measures of Interest

Several quantitative measures can be determined on the generated state space.

Measures and Importance Factors. Various measures are based on the *reliability function*, the cdf for the probability of a failure after a given time t. Another prominent measure is the *mean time to failure* (MTTF), the expected time until a system failure. The *variance* of the time to failure (VTTF) is obtained by $\mathrm{Var}(X) = E[X^2] - E[X]^2$ for random variable X, the time to failure. The *probability of failure* considers the limit probability of the reliability function for t to ∞. This is of interest as in DFTs not all events fail eventually, cf. Fig. 2(a). These measures can be used for single events in the DFT, and also for Boolean combinations of failed and operational gates, such as e.g., the expected time to a DFT state where events A and C have failed. Another measure-of-interest is the *expected number of faults before the DFT fails*; if this is high, it indicates that are various possibilities to take countermeasures. The *Fussell-Vesely importance factor* is the probability that a BE has failed when the DFT fails [17]. An exemplary *criticality importance factor* is the probability that a BE causes the DFT to fail. The measures above are analysed using efficient algorithms to verify temporal properties on CTMCs [18] or MA [19].

Conditional Measures. All measures except $R_F(t)$ can be conditioned on the occurrence of events, cf. the first column of Table 1. For example, as $\Pr_F \neq 1$ for some fault trees, the MTTF is not always defined. For this case a reasonable alternative is to condition the MTTF, assuming the DFT eventually fails.

Measure Preservation Under Optimisations. Techniques such as modularisation, DC-propagation and symmetry reduction are not applicable to all measures. Their robustness w.r.t. the measures is indicated in the last columns of Table 1, where $*$ means support of a light version. Modularisation is powerful

Table 1. Supported measures and importance factors

Symbol	Name	Cond.	Par.syn.	Mod.	dc.	Sym.red
$R_F(t)$	Reliability at t	✗	✗	✓	✓	✓
$\Pr_F$	Probability of failure	✓	✓	✓	✓	✓
MTTF_F	Mean time to failure	✓	✓	✗	✓	✓
VTTF_F	Variance of time to failure	✓	✓	✗	✓	✓
	Expected faults before failure	✓	✓	✗	✗	✓
	FV importance factor	✓	✓	✗	✗	$*$
	Criticality importance factor	✓	✓	✗	✓	$*$

if a partial state space suffices, e.g., if the measure is compositional –meaning that the measure can be obtained from its subDFTs' measures. This holds e.g., for reliability but not for MTTF. Symmetry reduction requires a lack of identity (of DFT nodes), which does not hold for some measures, including many conditional statements. If the lack of identity is not given, only a light version is applicable.

5 Parameter Synthesis

Problem. The analysis discussed so far has two drawbacks: It requires all failure rates in the DFT to be given and does not guarantee any robustness w.r.t. perturbations. The latter has been addressed by *sensitivity analysis* [9]. These deficiencies inspired us to treat *symbolic* failure rates, i.e. DFTs where failure rates and propagation probabilities in PDEPs are given as polynomials over a set of parameters (pDFTs). Our state space construction technique is largely unaffected by this. Our focus is on the failure rate synthesis in DFTs for any measure in Table 1, second column, i.e., *determine all values (of the symbolic rates) such that the DFT satisfies a given desired threshold on a measure*. For simplicity, we focus on DFTs that (after our reductions) obey no non-determinism, which applies to the vast majority of the DFTs in the literature. Thus, the underlying state space of pDFTs can be reduced to a *parametric* CTMC, i.e. a CTMC whose rates are polynomials over the DFT parameters.

Approach. To enable the synthesis in pDFTs we exploit the parameter synthesis tool `PROPhESY` [20]. Based on ideas in [15], it computes a closed form (precisely: a rational function) for a parametric CTMC and the measure of interest. To enable sensitivity analysis, it provides the derivative w.r.t. the parameters. On top of obtaining these functions, `PROPhESY` allows for parameter space partitioning — using satisfiability-modulo-theory (SMT) techniques for non-linear arithmetic. That is, given a pDFT, we can synthesise for which parameter values the measure (e.g., MTTF) is above a threshold. An example output is depicted

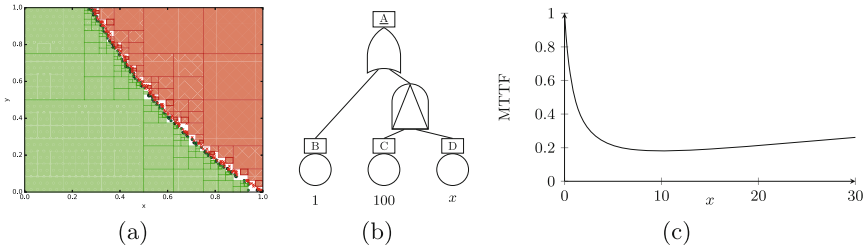


Fig. 5. (a) Sample output, (b) a sample parametric DFT, and (c) its MTTF. (Color figure online)

in Fig. 5(a). This plot was obtained for the DFT of Fig. 2(d) where W_1, W_2 and W_S have failure rates x, 1 and y respectively for unknown x, y.

The green boxes represent areas in which *all* failure rates of W_1 and W_S give rise to an MTTF that exceeds 1.5, while the red boxes guarantee *all* rates yield an MTTF below 1.5. For the white areas, none of the above statements can be made. Note that this output is extremely valuable as it provides information about many (in fact uncountably many) failure rate combinations for which the MTTF is below or above the threshold. We like to point out that obtaining this information is far from trivial, and intrinsically more involved than analysing a DFT where all failure rates are given. Consider the small example DFT from Fig. 5(b), where D has a symbolic failure rate. The MTTF of the DFT is given by the plot in Fig. 5(c). As the MTTF is not monotonic, the parameter synthesis is not straightforward.

6 Experiments

Set-Up. To evaluate our approach, we tested the performance of our tool on reliability and the MTTF assessment. We compare with the state-of-the-art tool `DFTCalc` [13] and assess the effect of our abstraction techniques. The experiments were conducted on an HP BL685C G7 restricted to 8 GB RAM and used a single 2.0 GHz core and a time-out of 1 h. We use the benchmark suite from [12]. Besides the smaller HCAS and SAP sets, it contains the following benchmarks:

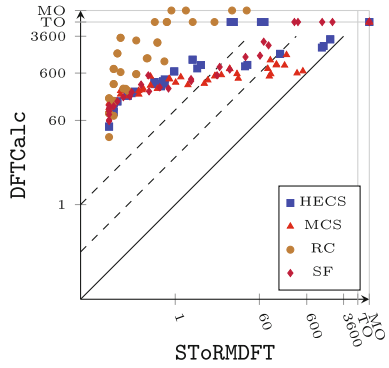
HECS. The *Hypothetical Example Computer System (HECS)* stems from the NASA handbook on FTs [2]. It features a computer system consisting of a processor, a memory unit (MU) and an interface consisting of hard- and software.

MCS. The *Multiprocessor Computing System (MCS)* contains computing modules consisting of a processor, a MU and two disks, the DFT was given in [10].

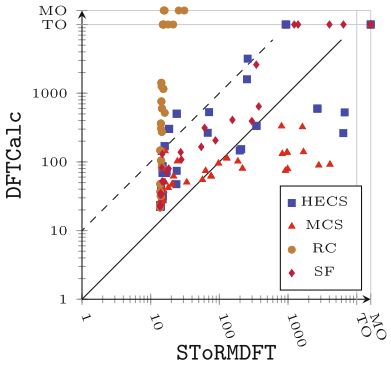
RC. The *Railway Crossing (RC)* is an industrial case modelling a level crossing which fails whenever any of the sensor-sets, barriers or controller fail [21].

SF. The *Sensor Filter (SF)* benchmark is a DFT that is automatically generated from an AADL (Architecture Analysis & Design Language) system model [22].

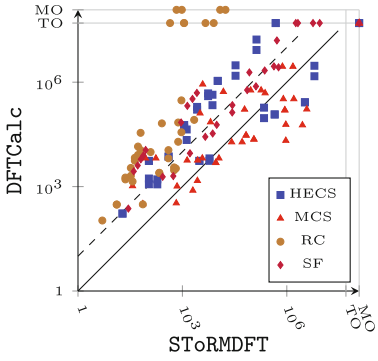
We used the simplified DFTs as produced in [12], as this is shown to be beneficial for `DFTCalc`. For each instance, we tested reliability for $t = 100$ and the MTTF. Further features were tested on a range of > 100 crafted instances.



(a) run time (seconds)



(b) memory footprint (MB)



(c) max. # states in MC

	reliability				MTTF			
	DFTCalc		STоRMDFT		DFTCalc		STоRMDFT	
	#	Time	#	Time	#	Time	#	Time
HECS(42)	38	2.8e4	42	3.1e0	36	2.6e4	40	7.0e3
MCS(42)	40	2.1e4	42	2.1e1	38	1.9e4	38	2.1e3
RC(38)	29	2.7e4	38	2.1e0	29	2.7e4	38	6.5e1
SF(30)	26	1.6e4	30	1.8e0	25	1.4e4	29	4.4e3
CAS(8)	8	1.3e3	8	3.6e-1	8	1.3e3	8	3.6e-1
SAP(4)	4	3.6e2	4	3.0e-1	4	3.2e2	4	1.6e-1

(d) #solved & total run time (seconds)

	reliability					MTTF			
	none	SR	DC	Mod	all	none	SR	DC	all
HC_2	30.3	15.6	1.1	0.05	0.04	29.9	15.5	1.2	0.61
MC_2	337.8	46.0	1.1	0.05	0.05	334.0	45.6	1.1	0.21
RC_{10}	53.6	0.1	53.5	0.20	0.05	53.9	0.1	53.6	0.07
SF_2^6	22.1	7.4	0.3	0.04	0.04	22.3	7.4	0.2	0.08

(e) optimisation run time (seconds)

Fig. 6. Overview of the experimental results on four different benchmark sets.

Results. Figures 6(a-c) compare the performance of our tool (referred to as `STоRMDFT`) with `DFTCalc` on MTTF (where modularisation is not applicable). All plots use a log-log-scale. Figure 6(a) presents the analysis time of a DFT. This includes state space generation. The lower dashed line indicates an advantage of our tool by a factor ten, the upper of a factor 100. The outer lines indicate TOs and MOs, respectively. Figure 6(b) indicates the peak memory consumption as given by the operating system. Figure 6(c) shows the peak intermediate state size. Figure 6(d) summarises the performance on the benchmark sets — it lists the number of benchmarks solved and the cumulative time needed for the solved benchmarks. Figure 6(e) shows the effect of the individual optimisation techniques (symmetry reduction, DC-propagation, modularisation) versus using all of them.

Observations. For non-parametric DFTs the performance is dominated by the state space construction. `STоRMDFT` creates intermediate state spaces that

are often ten times smaller; especially for moderately-sized DFTs, this is done with a much lower overhead. This results in generating state spaces up to 5 orders of magnitudes faster. The informed state space generation allows to stop exploring states where the measure of interest is settled. This advantage is best observed by comparing top events typed OR and AND. The former requires significantly smaller state spaces, which is reflected by the smaller intermediate state spaces — and leads to a significant advantage over `DFTCalc`. These effects are multiplied by aggressively applying symmetry reductions and DC-propagation. For many benchmarks, our abstractions directly yield the small bisimulation quotient. However, on some HECS and MCS instances, our symmetry reduction does not yet suffice and `DFTCalc` gains an advantage in terms of memory. Modularisation remains a powerful approach for assessing reliability. It profits additionally from the performance on small DFTs. Model-checking for reliability is for both `SToRMDFT` and `DFTCalc` so fast that our slightly better performance is hardly significant. For MTTF, `SToRMDFT` is significantly faster.

For parametric instances, the original DFTs from literature can be handled: For, e.g., the standard HECS from literature it takes 5 s to compute the rational function with more than 400 terms in the numerator. Parameter synthesis for 90 % of the parameter space finishes within four minutes. However, scalability beyond these moderately-sized DFTs remains an open issue, as the parameters appear throughout the full state space.

7 Conclusions and Future Work

We have presented a state space generation technique for DFTs that is more than two orders of magnitude faster than the state-of-the-art. The technique is complemented with a new feature in DFT analysis — the synthesis of failure rates for measures such as MTTF. Future work includes the failure rate synthesis for reliability (e.g., using [23]) and improve scalability for parameterised MTTF.

Acknowledgement. We thank Christian Dehnert for fruitful discussions. This work was supported by the Excellence Initiative of the German federal and state government, the CDZ project CAP (GZ 1023), and the BMBF project HODRIAN.

References

1. Dugan, J.B., Bavuso, S.J., Boyd, M.: Fault trees and sequence dependencies. In: Proceedings of RAMS, pp. 286–293 (1990)
2. Stamatelatos, M., Vesely, W., Dugan, J.B., Fragola, J., Minarick, J., Railsback, J.: Fault Tree Handbook with Aerospace Applications. NASA Headquarters, Washington, D.C. (2002)
3. Coppit, D., Sullivan, K.J., Dugan, J.B.: Formal semantics of models for computational engineering: a case study on dynamic fault trees. In: Proceedings of ISSRE, pp. 270–282 (2000)
4. Sullivan, K., Dugan, J.B., Coppit, D.: The Galileo fault tree analysis tool. In: Proceedings of FTCS, pp. 232–235 (1999)

5. Boudali, H., Crouzen, P., Stoelinga, M.I.A.: A rigorous, compositional, and extensible framework for dynamic fault tree analysis. IEEE Trans. Dependable Secure Comput. **7**(2), 128–143 (2010)
6. Junges, S., Guck, D., Katoen, J.P., Stoelinga, M.: Uncovering dynamic fault trees. In: Proceedings of DSN (2016, to appear)
7. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)
8. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
9. Ou, Y., Dugan, J.B.: Sensitivity analysis of modular dynamic fault trees. In: Proceedings of IPDS, pp. 35–43 (2000)
10. Montani, S., Portinale, L., Bobbio, A., Codetta-Raiteri, D.: Automatically translating dynamic fault trees into dynamic Bayesian networks by means of a software tool. In: Proceedings of ARES, pp. 804–809 (2006)
11. Walker, M., Papadopoulos, Y.: Qualitative temporal analysis: towards a full implementation of the Fault Tree Handbook. Control Eng. Pract. **17**(10), 1115–1125 (2009)
12. Junges, S., Guck, D., Katoen, J.P., Rensink, A., Stoelinga, M.: Fault trees on a diet - automated reduction by graph rewriting. In: Li, X., Liu, Z., Yi, W. (eds.) SETTA 2015. LNCS, vol. 9409, pp. 3–18. Springer, Heidelberg (2015)
13. Arnold, F., Belinfante, A., Van der Berg, F., Guck, D., Stoelinga, M.: DFTCalc: a tool for efficient fault tree analysis. In: Bitsch, F., Guiochet, J., Kaâniche, M. (eds.) SAFECOMP. LNCS, vol. 8153, pp. 293–301. Springer, Heidelberg (2013)
14. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: Proc. of LICS, pp. 342–351. IEEE Computer Society (2010)
15. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005)
16. Gulati, R., Dugan, J.B.: A modular approach for analyzing static and dynamic fault trees. In: Proceedings of RAMS, pp. 57–63 (1997)
17. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15–16**, 29–62 (2015)
18. Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. **29**(6), 524–541 (2003)
19. Guck, D., Hatefi, H., Hermanns, H., Katoen, J.P., Timmer, M.: Analysis of timed and long-run objectives for Markov automata. LMCS **10**(3), 17 (2014)
20. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruintjes, H., Katoen, J.-P., Ábrahám, E.: PROPhESY: a PRObabilistic ParamEter SYnthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Heidelberg (2015)
21. Guck, D., Katoen, J.P., Stoelinga, M., Luiten, T., Romijn, J.: Smart railroad maintenance engineering with stochastic model checking. In: Proceedings of RAILWAYS, Civil-Comp Proceedings, Civil-Comp Press, vol. 104, pp. 299–314 (2014)
22. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**, 754–775 (2011)
23. Češka, M., Dannenberg, F., Kwiatkowska, M., Paoletti, N.: Precise parameter synthesis for stochastic biochemical systems. In: Mendes, P., Dada, J.O., Smallbone, K. (eds.) CMSB 2014. LNCS, vol. 8859, pp. 86–98. Springer, Heidelberg (2014)