Smart Semantics for Fault Trees

Jip Spel
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.j.spel@student.utwente.nl

ABSTRACT

Systems used e.g. in nuclear power plants and railroad infrastructures require analysis to know about the risk of failures and the effect of different maintenance strategies on this risk. One way to analyse these systems is by building Dynamic Fault Trees (DFTs) and analysing the corresponding Input/Output Interactive Markov Chains (I/O-IMCs). Although the I/O IMCs are used for analysis, there is a major drawback, namely the size of the state space. For larger systems this state space becomes too large to do any further calculations. Therefore, analysis of larger FTs is limited by their size.
During the conversion from DFTs to I/O-IMCs irrelevant behaviour arises. Therefore, we present smart semantics for fault trees, by applying context-dependent state space generation. This includes an algorithm which sets a boolean to determine the type of the context. This boolean determines whether application of these smart semantics is feasible. Furthermore, we propose a definition of irrelevant behaviour which arises in the conversion of a DFT into an I/O-IMC.

Keywords

context-dependent, DFT, DFTCalc, I/O-IMC, irrelevant behaviour, maintenance, risk analysis, smart semantics, state space

1. INTRODUCTION

Risk analysis is crucial for safety critical systems like nuclear power plants, railroad systems and medical equipment. Failure of these systems can be life threatening, therefore it is desirable to minimise this failure. Furthermore, e.g. a ticket system or the lottery require risk analysis, since failure will lead to high costs.

One way to perform risk analysis is Fault Tree Analysis (FTA). Fault Trees (FTs) provide a way to structurally analyse the impact of the failure of a component on the system, in terms of availability and reliability. Furthermore, some FTAs integrate maintenance [11] which provides a way to incorporate the influence of maintenance on the failure rates.

Standard FTs are limited in their applicability, since they don't consider the sequence of failure. Therefore, Dynamic Fault Trees (DFTs) [8] were introduced. They extend standard FTs by including additional dynamic gates, which extend the usability of FTA. In the remainder of this paper, we will refer to standard FTs as FTs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
23th *Twente Student Conference on IT* June 22th, 2015, Enschede, The Netherlands.
Copyright 2015, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

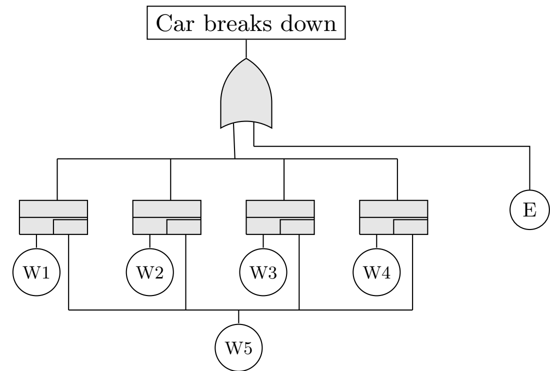


Figure 1: Example of a DFT

EXAMPLE 1. *Figure 1 shows a simplified DFT for a car. The circles are basic events, they represent elements of the car. The rectangular shapes are spare gates, they have a primary input and an input for a spare component. In this example the four wheels share one spare wheel. This spare wheel will be used when one of the other wheels fail. However, when the spare wheel is in use and another wheel fails, the car breaks down. Furthermore, as modelled by the OR-gate the car will break down if the engine fails.*

One way to analyse DFTs is with continuous-time Markov chains (CTMCs). CTMCs model every state of the components of the DFT. Since a component in a non-maintainable DFT is in an inactive, active or failed state, the state space is exponential to the number of basic events.

Using input/output interactive Markov chains (I/O-IMCs) instead of CTMCs partly relieves [4] the drawback of the exponential state space, since it allows intermediate minimisation and hiding. Furthermore, this analysis with I/O-IMCs is fully compositional [4] which allows modular analysis of the DFT. DFTCalc is a tool for Fault Tree Analysis (FTA) which uses I/O-IMCs instead of CTMCs [1]. DFTCalc transforms each component of the DFT into an I/O-IMC with the help of module templates.

EXAMPLE 2. *Figure 2 shows an I/O-IMC for a basic event in a system. Several transitions may occur in this I/O-IMC, a? activates the component, λ indicates the failure rate of a component. Furthermore, the component can fail when inactive, μ indicates this failure rate.*

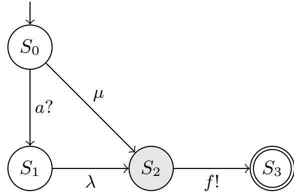


Figure 2: Example of an I/O-IMC

Problem Statement.
Even though using I/O-IMCs with intermediate minimisation and hiding instead of CTMC shows a serious state space reduction [4], the state space still is the bottleneck in analysing FTs. Therefore, additional state space reduction is desirable. This research sets out to define smart semantics for fault trees by applying context-dependent state space generation, which includes removing unnecessary behaviour to reduce the state space. Therefore, we consider the following question:

> How to apply context-dependent state space generation in the module templates of DFTCalc?

To answer this question we analyse the following subquestions:

1. Which contexts can be found in the conversion from a DFT to an I/O-IMC?
2. To what extend does context-depent state space generation in DFTCalc cause state space reduction in terms of the size of the state space and the computation time?
3. How to redesign the module templates of DFTCalc for the different contexts?
4. How to detect the context for the state space generation in DFTCalc?

When it is possible to detect different contexts in DFTCalc and to apply context-dependent state space generation in the module template, we will implement the application of these smart semantics in the tool DFTCalc.

Related Work.
In 1981 one of the first handbooks about fault trees was written by Vesely et al. [18]. It was developed to document material on fault tree construction and evaluation. Dugan et al. [8] introduced the additional gates of DFTs to be able to model sequence dependencies in FTs. In 2002 the NASA [17] presented a new version of the handbook which also includes these dynamic gates.

Also research on the formalisation of FTs, extentions to Markov Chains and the relation between these two have been done. Hermanns et al. [12] describe Interactive Markov Chains (IMCs) and explain the necessity. Dugan et al. [9] introduced a formalisation of DFT's. Boudali et al. [3] describe how DFTs can be analysed by using I/O-IMCs. They also formalise DFTs and show how the elements of a DFT can be transformed into an I/O-IMC [2]. A technique to integrate maintenance in FTA is presented by Guck et al. [11], they also introduce smart semantics for failure and repair of elements by pattern matching in the FTs, finding equivalent behaviour of elements and aggregation of similar BEs.

Besides the formalisation research, also research on the analysis of FTs can be found. Patterson-Hine et al. [15] developed a modular analysis of dynamic fault trees.
Boudali et al. provide a framework for DFT analysis [4]. Arnold et al. [1] provide a tool, DFTCalc, for analysing DFTs with I/O-IMCs.

Ruijters et al. [16] provide an overview of the research done on FTs, DFTs and how to analyse them. Furthermore, Junges' Master Thesis [13] describes when DFTs are well-formed.

Organisation of the paper.
The remainder of this paper is organised as follows. Section 2 discusses FTs, DFTs and I/O-IMCs and introduces their formal definition. Furthermore, this section shows the transformation from a DFT to an I/O-IMC. Section 3 gives the definition of behaviour in I/O-IMCs and describes the different contexts of behaviour. Furhtermore, section 3.2 shows how the different context-types arise in a DFT. In section 4 we state our approach to the problem. Section 5 contains the results which we obtained during the research process. We discuss these results in section 6 and we conclude in section 7. Furthermore, we propose future work in section 8.

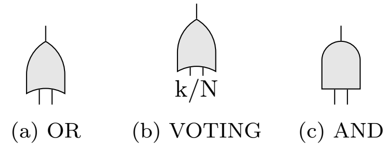


Figure 3: The gates of a FT

2. BACKGROUND

2.1 Fault Trees

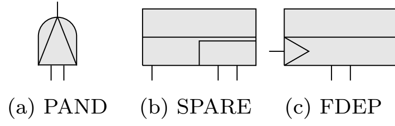
FTs consist of basic events (BE), which model the failure of a physical component of the system, and logical gates. Figure 3 shows the following gates, which exist in FTs:

- **OR** The OR gate fails if one of its inputs fails.
- **k/N** The k/N gate is also known as a **VOTING** gate, which fails when k of the N inputs fail. A well known k/N gate is the 1/N gate, which is an OR gate.
- **AND** The AND gate fails if all of its inputs fail.
- **INHIBIT** The INHIBIT gate will forward the failure when all of its inputs fail, and the additional event also occurs. The INHIBIT gate is basically the same as an AND gate and is therefore not considered in the rest of this paper.

FTs are directed acyclic graphs (DAGs). FTs disregard the failure sequence, since they only consider the combination of failure.

2.2 Dynamic Fault Trees

DFTs extend FTs by having dynamic gates. These gates enable the possibility to regard the sequence of failure and the usage of spare elements. When in figure 1 the fifth



(a) PAND (b) SPARE (c) FDEP

Figure 4: The dynamic gates of a DFT

wheel breaks down, the car will still work, since this is a spare wheel. However, FTs can't model the usage of spare elements. Figure 4 shows the additional gates of a DFT, which are listed below.

- **PAND** The PAND gate will forward failure if all of its inputs fail in the correct order, which is from left to right.
- **SPARE** The SPARE gate allows an additional component to be added to the system. It consists of a primary input and zero or spare elements.
- **FDEP** The functional dependency (FDEP) gate consists of a trigger event and one or more dependent events. If the event triggers all dependent events will fail.

Furthermore, a DFT has, as a FT, BEs. BEs are inactive, active or failed. When the BE is active, λ denotes its failure rate. Additionally, BEs can include a parameter α, the dormancy factor. This dormancy factor is between 0 and 1 and allows the BE to have a different failure rate when inactive, namely failure rate $\mu = \alpha\lambda$. There are two special cases, when α is 0 the BE is a cold spare, in this case the failure rate when inactive is 0. When α is 1 the BE is a hot spare, the failure rate is the same as when the BE is inactive and active.

Definition 4 in [2] formalises the syntax of the elements of a DFT. Furthermore, definition 5 in [2], as cited below, formalises the syntax of the DFT.

DEFINITION 1. *A dynamic fault tree is a triple $D = (V, preds, l)$, where*

– *V is a set of vertices,*

– *$l : V \to E$ is a labeling function, that assigns to each vertex a DFT component,*

– *preds: $V \to V^*$ is a function that assigns to each vertex a list of inputs.*

Furthermore, Junges [13] proposed a definition for well-formed DFTs. In this paper we assume that all DFTs which are analysed with context-dependent state space generation are well-formed.

2.3 Maintenance of FTs

The maintenance of FTs includes inspections, repairs, renewals and spare management of the system. The behaviour of the BEs is therefore extended with repair possibilities. Furthermore, a repair module is introduced which listens for the failure of the BE. When a BE fails, this module sends out a repair request and subsequently sends the BE an up signal after a successful repair [11]. Furthermore, maintenance introduces a repair unit which handles the order of repair of the elements.

2.4 I/O-IMCs

An I/O-IMC consists of states, input actions, output actions, internal actions and Markovian transitions. An input action (notated with ?) requires synchronisation on an output action (notated with !). Therefore, input actions can only be taken when the output action occurs. Internal actions happen immediately and do not require synchronisation on other I/O-IMCs. Markovian transitions, labelled with λ and μ, represent system delay.

Definition 1 in [4], as cited below, contains the formalism of I/O-IMCs.

DEFINITION 2. *An input/output interactive Markov chain P is a tuple $< S, s^0, A, \to, \to^M >$, where*

– *S is a set of states,*

– *$s^0 \in S$ is the initial state.*

– *A is a set of discrete actions (or signals), where $A = (A^I, A^O, A^{int})$ is partitioned into a set of input actions A^I, output actions A^O and internal actions A^{int}. We write $A^V = A^I \cup A^O$ for the set of visible actions of P. We suffix input actions with a question mark (e.g. a?), output actions with an exclamation mark (e.g. a!) and internal actions with a semi-colon (e.g. a;).*

– *$\to \subseteq S \times A \times S$ is a set of interactive transitions. We write $s \xrightarrow{a} s'$ for (s, a, s') $\in \to$. We require that I/O-IMCs are input-enabled: $\forall s \in S, a? \in A^I, \exists s' \in S \cdot s \xrightarrow{a} s'$*

– *$\to^M \in S \times \mathbb{R}_{>0} \times S$ is a set of Markovian transitions. We write $s \xrightarrow{\lambda}{}^M s'$ for (s, λ, s) $\in \to^M$*

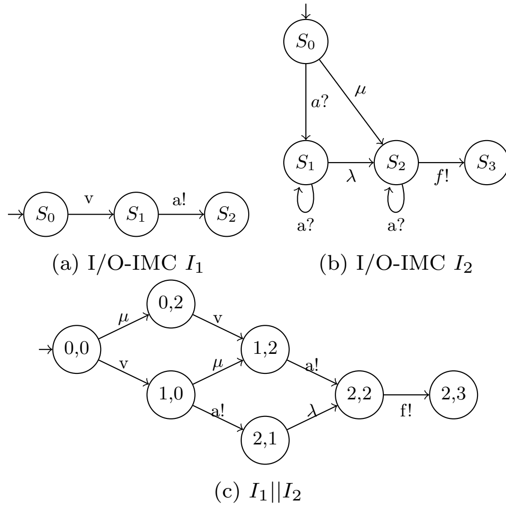
We denote the elements of P by $S_P, S^0_P, A_P, \to_P, \to^M_P$ and omit the subscript P whenever clear from the context. The action signature of an I/O-IMC is the partitioning (A^I, A^O, A^{int}) of A. We denote the class of all I/O-IMCs by IOIMC.

2.4.1 Parallel composition

One of the properties of I/O-IMC is that they are compositional, this means that a set of I/O-IMCs which correspond to elements of a system can be aggregated into one one I/O-IMC. This aggregation is done in a stepwise and hierarchical way [1]. Arnold et al. denote parallel composition of two I/O-IMCS I_1 and I_2 with $I_1||I_2$. We will also use this notation. The result I_c of $I_1||I_2$ is also an I/O-IMC and it has as state space the Cartesian product of the state spaces of I_1 and I_2. Furthermore the following two rules are used (as stated in [1]):

- If an action doesn't require synchronization, then I_1 and I_2 evolve independently.
- If an action a? on an interactive transition requires synchronisation, then it can only be taken at the time when another I/O-IMC performs output on a!.

EXAMPLE 3. *Figure 5 shows the parallel composition of two I/O-IMCs. Where I/O-IMC I_1 shows an activation of a component after a delay denoted by v, I/O-IMC I_2 shows a BE of a DFT, which can be activated with the activation signal. In the parallel composition of I_1 and I_2 the states are denoted by first the corresponding state in I_1 and secondly the state in I_2. Furthermore, there is synchronisation on action a.*



(a) I/O-IMC I_1 (b) I/O-IMC I_2

(c) $I_1||I_2$

Figure 5: Parallel composition of two I/O-IMCs

2.5 From DFT to I/O-IMC

DFTCalc builds a minimal I/O-IMC to analyse the DFT [1]. The creation of an I/O-IMC from a DFT involves the following steps.

1. Each component is transformed into an I/O-IMC, with the help of module templates.
2. Two I/O-IMCs are parallel composed to one I/O-IMC, a heuristic determines which IMCs will be composed.
3. The signals that are no longer necessary for composition are hidden.
4. The resulting I/O-IMC is minimised.
5. When there are still two or more I/O-IMCs left, move back to step 2.

Step 1 transforms each component in the DFT into an I/O-IMC. Every I/O-IMC has an initial state (indicated with an incoming arrow), intermediate states, a failed state (indicated with grey) and an absorbing state (indicated with an additional circle).

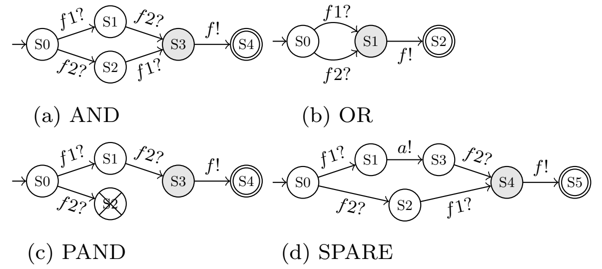
Definition 6 in [2] gives the relationship between a DFT and I/O-IMC. This definition leads to the following I/O-IMCs corresponding to gates:

AND Figure 6.a shows how an AND gate with two inputs can be transformed into an I/O-IMC.

Basic Events Figure 2 shows the I/O-IMC belonging to a BE with dormancy factor α, such that $\mu = \alpha\lambda$. When α is between 0 and 1, the BE is a warm BE. When α is 0, the BE is a cold BE and the transition from state s0 to state s2 can be removed. When α is 1, the BE is a hot BE, figure 2 could have been simplified by removing state s1, since the failure rate would always be λ, independent of whether the component is active or inactive.

FDEP The functional dependency (FDEP) is modelled with a firing auxiliary function FA [2]. It can be seen as an OR port where a firing signal is sent when one of the dependent functions fails or when any of the trigger events occur.

OR Figure 6.b shows the I/O-IMC corresponding to an OR gate with 2 input events, if one of the input events is triggered, the firing signal is sent.



(a) AND (b) OR

(c) PAND (d) SPARE

Figure 6: I/O-IMCs for non-maintainable gates in a DFT

PAND The priority AND gate with two inputs corresponds to figure 6.c. When f2? fails first the system will move to an absorbing state, indicated with X.

SPARE The I/O-IMC for a spare gate with 1 primary and 1 spare component corresponds to figure 6.d. When the input fails an activation signal is sent, which activates a spare component. When two subsystems have the same spare component a they need to communicate when they use a. When the first subsystem uses a the second subsystem receives the input action, a?, hence it knows that the spare component is no longer available.

VOTING The OR gate can also be seen as an 1/2 voting gate, when k of the n inputs fail, the failure will be propagated.

3. DIFFERENT CONTEXTS

To apply context-dependent state space generation, it is necessary to determine in which contexts a different state space generation is necessary. In the conversion of a DFT into an I/O-IMC a context distinction can be made between:

- Type 1: elements of the DFT which are known to be activated directly when the system starts and can't get to an inactive state.
- Type 2: elements of the DFT which will be inactive for some time, or which might become inactive.

The structure of the DFT shows the context of a component is, based on definitions 4, 5 and 6.

When it is known from the structure of the DFT that the component is of context type 1, irrelevant behaviour arises. This irrelevant behaviour consists of two parts. First of all the activation transition will be superfluous, since it is known that the component will be activated right at the start of the system. Secondly, the transition which models failure while the component is inactive is unnecessary, since the component is activated directly at the start. The following definitions will formalise these statements. Hereby, we will use the definitions of a DFT and an I/O-IMC as stated in section 2.

3.1 Definitions

First of all, we consider behaviour in an I/O-IMC as all the possible transitions. So it is the union of the interactive and Markovian transitions.

Definition 3. *In an I/O-IMC behaviour is a transition* $s_i \xrightarrow{x} s_j \in \rightarrow \cup \rightarrow^M$ *where* $x \in A \cup \mathbb{R}_{>0}$. *All the behaviour in an I/O-IMC forms* $\rightarrow \cup \rightarrow^M$

To apply context-dependent state space generation, it is necessary to know what behaviour is irrelevant in an I/O-IMC.

Definition 4. *Behaviour* $b_1 = s_i \xrightarrow{x} s_j$ *in I/O-IMC M is irrelevant when*

- b_1 *will occur directly when the I/O-IMC starts or*
- $\exists$ *behaviour* $b_2 = s_i \xrightarrow{x} s_k$ *where* b_2 *will occur before* b_1 *or*
- b_1 *will never occur.*

This previous definition only states when behaviour is irrelevant. However, we need a formalism for irrelevant behaviour in the conversion of a DFT to an I/O-IMC. Since we distinct two types of DFTs namely, non-maintainable and maintainable, we also need two definitions namely, definitions 5 and 6.

Definition 5. *Behaviour* $b_1 = s_i \xrightarrow{x} s_j$ *in I/O-IMC M is irrelevant when in the corresponding non-maintainable DFT D = (V, preds, l)*

- b_1 *corresponds to the activation part of* $v \in V$ *and,*
- *l(v) has no spare parents in D or p is a spare parent of l(v)* $\rightarrow$ *l(v) is primary input in preds(p).*

or when,

- $\exists$ *behaviour* $b_2 = s_i \xrightarrow{x} s_k$ *and,*
- b_2 *corresponds to the activation part of* $v \in V$ *and,*
- *l(v) has no spare parents in D or p is a spare parent of l(v)* $\rightarrow$ *l(v) is primary input in preds(p).*

or when,

- $\forall$ *behaviour* $b_2 = s_h \xrightarrow{x} s_i$ b_2 *is irrelevant.*

Definition 6 differs from definition 5 by handling the children of spare elements differently. After a failed spare is repaired, it should change from active to inactive, so even though the primary spare was activated directly at the start of the system, it might eventually get into an inactive state.

Definition 6. *Behaviour* $b_1 = s_i \xrightarrow{x} s_j$ *in I/O-IMC M is irrelevant when in the corresponding maintainable DFT D = (V, preds, l)*

- b_1 *corresponds to the activation part of* $v \in V$ *and,*
- *l(v) has no spare parents in D.*

or when,

- $\exists$ *behaviour* $b_2 = s_i \xrightarrow{x} s_k$ *and,*
- b_2 *corresponds to the activation part of* $v \in V$ *and,*
- *l(v) has no spare parents in D.*

or when,

- $\forall$ *behaviour* $b_2 = s_h \xrightarrow{x} s_i$ b_2 *is irrelevant.*

Example 4. *Figure 7 shows the I/O-IMC corresponding to node A. Since node A is a basic event and the gate is an OR gate connected to the top event, it is known that A will always be active. The corresponding I/O-IMC also shows the activation transition and a probability of failure while inactive. These two parts of behaviour are irrelevant.*

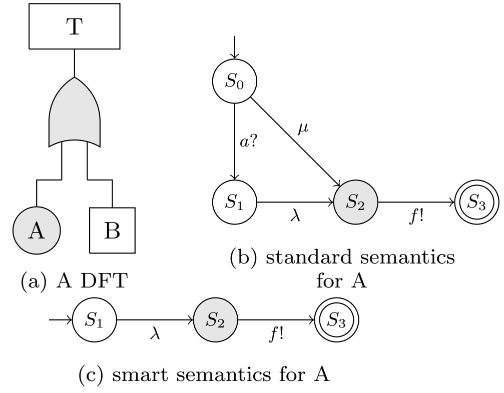


Figure 7: An example of irrelevant behaviour

Example 5. *Figure 8 shows the I/O-IMC corresponding to the DFT in the figure. Since there are no spare elements in this DFT, and the AND gate is connected to the top level event, it is known that the AND gate is activated directly at the start of the system. Furthermore, A and B are children of the AND gate, so they will also be activated directly at the start of the system. So, this whole DFT is of the first context-type. When we parallel compose the I/O-IMCs of the elements of the DFT, we will get figure 8.a. Since the whole system will be activated directly at the start the following transitions will never occur:* $s_0 \xrightarrow{\mu_b} s_1,\ s_0 \xrightarrow{\mu_a} s_2,\ s_1 \xrightarrow{\mu_a} s_{12},\ s_2 \xrightarrow{\mu_b} s_{12},\ s_4 \xrightarrow{\mu_b} s_3,\ s_5 \xrightarrow{\mu_a} s_6,\ s_7 \xrightarrow{\mu_b} s_{12}$ *and* $s_9 \xrightarrow{\mu_a} s_{12}$.

Furthermore, the input activation signals can be removed, which are:

$s_0 \xrightarrow{a?} s_4,\ s_0 \xrightarrow{b?} s_5,\ s_1 \xrightarrow{a?} s_3,\ s_2 \xrightarrow{b?} s_6,\ s_4 \xrightarrow{b?} s_8,\ s_6 \xrightarrow{a?} s_8,\ s_7 \xrightarrow{b?} s_{10}$ *and* $s_9 \xrightarrow{a?} s_{11}$.

This removal leads to figure 8.c. When λ_a *equals* λ_b *this I/O-IMC will be further optimised into 8.d*

3.2 Different types in DFT

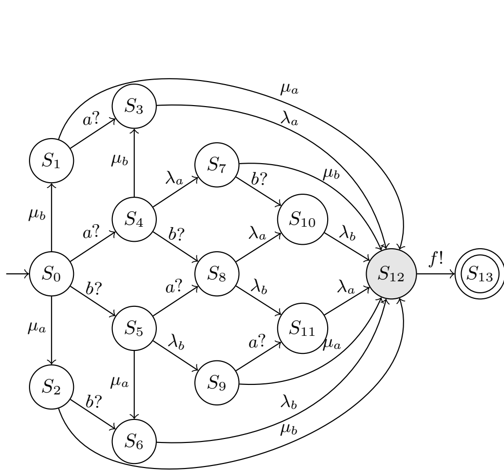
These definitions together with the two types lead to five different situations:

1. non-maintainable component with no spare parents;
2. non-maintainable component which is the primary input of the spare parent;
3. non-maintainable component with spare parents;
4. maintainable component with no spare parents;
5. maintainable component with spare parents.

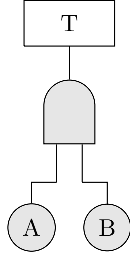
Numbers 1, 2 and 4 are of context-type 1, number 3 and 5 are of context-type 2. Since a maintainable component which is the primary inputs of a spare may become inactive, all inputs for a maintainable spare will be of type 2. Figure 9 shows the I/O-IMCs corresponding to BEs of the different situations.

4. APPROACH

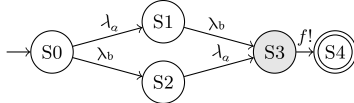
This section discusses the steps we took to analyse the problem and to achieve the goal of reducing the state space by applying smart semantics.



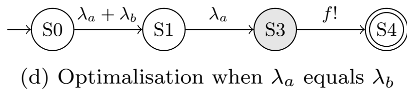
(a) Standard semantics for an DFT consisting of an AND gate with two BEs



(b) DFT with an AND gate

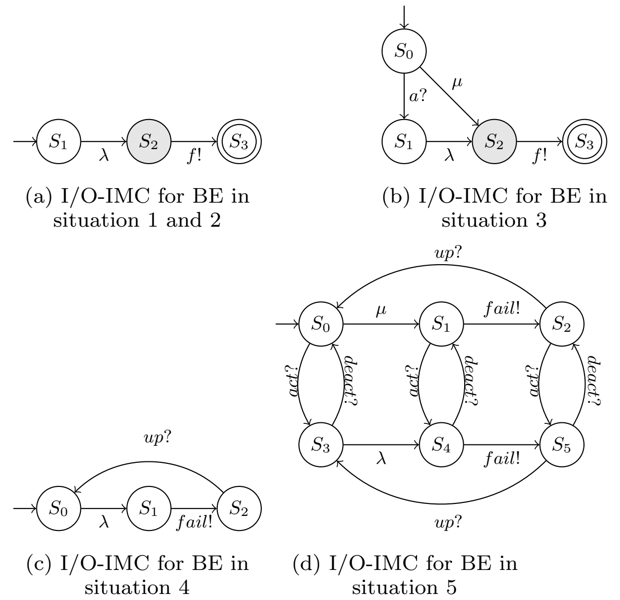


(c) Smart semantics for an DFT consisting of an AND gate with 2 BEs



(d) Optimalisation when λ_a equals λ_b

Figure 8: An example of irrelevant behaviour in the conversion of an AND gate



(a) I/O-IMC for BE in situation 1 and 2

(b) I/O-IMC for BE in situation 3

(c) I/O-IMC for BE in situation 4

(d) I/O-IMC for BE in situation 5

Figure 9: The I/O-IMCs corresponding to different types of BEs.

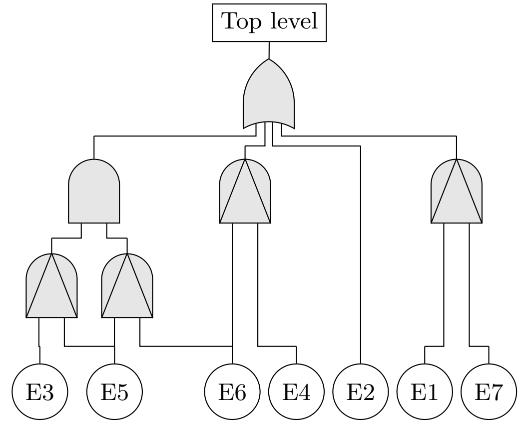


Figure 10: DFT for the sensor filter

In section 3 we answer the first subquestion by giving the definition behaviour and determining the different contexts in which state spaceare generated. To determine different contexts in DFTCalc we analysed several case studies in which we transformed DFTs into I/O-IMCs. Section 4.1 contains these case studies. Furthermore, we studied the source code of DFTCalc, to understand the module templates which are used in the conversion of a DFT to an I/O-IMC.

After the determination of the different contexts, we analysed the effect of applying context-dependent state space generation. We did this by comparing the final and maximal state space and computation time of I/O-IMCs of different DFTs with and without the smart semantics. We used these results to determine that the smart semantics contribute significant to the state space reduction.

Since the state space reduction through smart semantics is satisfactory, we implemented the smart semantics in DFTCalc. Furthermore, we constructed an algorithm to detect when we are in a different context. This algorithm checks whether or not a component has spare parents, which depends the context in which the state space needs to be generated.

4.1 Case studies

To analyse the effect of the removal of the irrelevant behaviour we chose five case studies:

- the cardiac assist system (CAS) [6];
- the cascaded PAND system (CPS) [6, 5];
- the fault-tolerant parallel processor(FTPP) [10];
- the multiprocessor computing system (MCS) [14];
- the sensor filter (SF) [7].

CAS, CPS and FTPP are further specified in [2], in [1] a further specification of MCS can be found. The SF is an adaptation of [7]. We chose these five case studies since, the CAS, FTPP and MCS all contain one or more spare

gates, hence, smart semantics is not applicable for all elements of the DFT. The CPS and SF however, contain no spare gates, therefore all basic events are active.

All of the case studies were held on a virtual machine with 2 processors and a base memory of 2048 MB.

Cardiac assist system (CAS)
The CAS consists of three independent modules, namely the CPU, the motors and the pumps. If one of these modules fails, the system will fail.

Cascaded PAND system (CPS)
The CPS consists of three identical independent modules. These modules consist of four elements and will only fail when all of the elements fail. The sequence in which these modules fail does matter, a PAND gate models this behaviour. All the BEs in the CPS are warm.

Fault-tolerant parallel processors (FTPP)
The FTPP-n consists of four logical groups of n processors. Each group has a shared cold spare. A network component physically connects one processor of each group. When this network component fails all connected elements will be unavailable. In this paper the FTPP-4 is reffered to as FTPP.

Multiprocessor computing system (MCS)
The MCS consists of two computing modules which are connected via a bus. Furthermore, they are powered by a power supply and they share a memory module. A computing module consists of a processor, a memory, a hard drive and a spare hard drive. The MCS will fail when the bus fails or when a computing module fails. A computing module will fail when the power supply fails, when both of its hard drives fail or when its memory fails and the spare memory module is already in use or already failed.

Sensor Filter (SF)
The SF consists of AND and PAND gates, which are connected to the top level through an OR gate. Furthermore, all BEs have a dormancy factor of 0. Figure 10 shows a DFT for SF.

5. RESULTS

Section 3 answers the first subquestion by stating that there are two different context-types for which context-dependent state space generation is necessary, namely the first type, in which the component of the DFT is directly activated at the start of the system, and the second type in which the component of the DFT might be inactive for some time at the start of the system. This section shows the other results of this research by stating an algorithm to detect different context, giving the steps of the implementation of context-dependent state space generation in DFTCalc, and providing the final results on the state space reduction.

5.1 Algorithm

To apply context-dependent state space generation, we constructed an algorithm to determine which context-type we are considering. This algorithm is based on definitions 5 and 6. It consists of two parts, a begin and a recursive part. The first part sets all elements of the DFT to not initialised, and starts the initialisation by calling initialise() on the top level node.

The initialisation is the recursive part of the algorithm. Two versions of this initialisation exist, the first one, procedure 2, only considers non-maintainable DFTs with the following gates: AND, OR, PAND, POR, SPARE and FDEP. The second one, procedure 3 is applicable for maintainable DFTs.

Procedure 1 Begin detection context-type

```
1: for all elements of DFT do
2:     component.initialised ← false
3: end for
4: begin ← DFT.getTopNode()
5: begin.initialise()
```

Procedure 2 considers non-maintainable DFTs. When a component is not initialised, it is known that it has no spare parents, since this algorithm works recursively. We hereby assume that the DFTs are well defined, as stated in definition 4.9 of [13] When the component type is spare, it will initialise all its spare children to not active at the start. Furhtermore, when a component itself is not activated at the start of the system, it will set all of its children to not activated as well.

Procedure 2 initialise() for a non-maintainable DFT

```
1: if not component.initialised then
2:     component.active ← true
3:     component.initialised ← true
4: end if
5: if component.type is SPARE then
6:     for all sparechildren of component do
7:         child.active ← false
8:         child.initialised ← true
9:     end for
10: end if
11: if not component.active then
12:     for all children of component do
13:         child.active ← false
14:         child.initialised ← true
15:     end for
16: end if
17: for all children of component do
18:     child.initialise();
19: end for
```

Procedure 3 considers maintainable DFTs. The only difference between this procedure and procedure 2 is that when a component type is SPARE it will initialise all its children to not activated directly at the start of the system. Since a primary input of a SPARE gate in a maintainable DFT can become inactive.

Procedure 3 initialise() for a maintainable DFT

```
1: if not component.initialised then
2:     component.active ← true
3:     component.initialised ← true
4: end if
5: if component.type is SPARE or not component.active
   then
6:     for all children of component do
7:         child.active ← false
8:         child.initialised ← true
9:     end for
10: end if
11: for all children of component do
12:     child.initialise();
13: end for
```

Table 3: Computation time for toolchain with MRMC

	Average time		**Reduction**
Model	**original**	**smart semantics**	
CAS	0m38.98s	0m36.90s	5.33%
MCS	0m39.69s	0m37.35s	5.89%
FTPP	2m36.45s	2m19.25s	10.99%
CPS	0m28.61s	0m26.14s	8.64%
SF	0m24.07s	0m23.35s	2.99%

5.2 Implementation

For the implementation of context-dependent state space generation in DFTCalc we implemented the algorithm to determine the context-type of the component of the DFT. Furthermore, we added special module templates for the first context-type.

5.2.1 Algorithm in DFTCalc

We implemented the algorithm with the initialisation as in procedure 2 in DFTCalc for non-maintainable DFTs. However, with a slight adaptation to this initialisation part, as stated in initialisation procedure 3 it can be extended to work for maintainable DFTs as well.

5.2.2 Additional module templates

DFTCalc uses module templates to transform each component of a DFT into an I/O-IMC. These templates also state the case in which a component is activated, and the transition for failure while inactive. With the application of context-dependent state space generation these two situations are unnecessary , therefore, we added module templates for the first context-type.

5.3 Case Studies

We implemented the application of smart semantics for all basic gates as described in 2. The results of the case studies can be found in table 1. Furthermore, we calculated the percentage of final state space reduction and maximal state space reduction for the case studies. These results can be found in table 2. Additionally, table 3 contains the computation time of the entire toolchain.

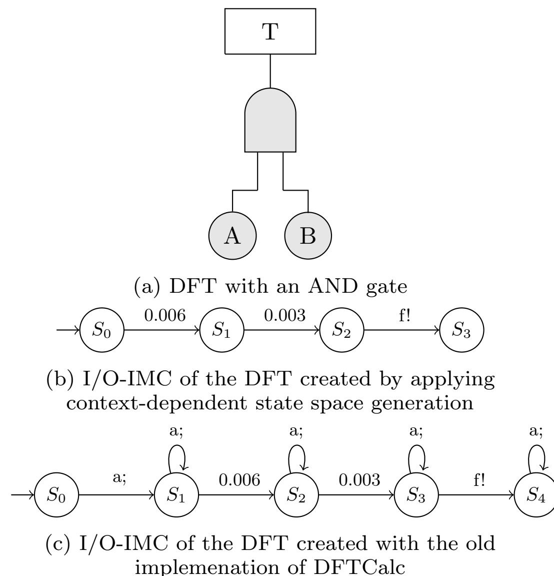
The probability of failure for all five case studies with and without the smart semantics is the same, furthermore, no errors occur during compilation of DFTCalc and computation of the DFTs.

6. DISCUSSION

6.1 Results

All of the five case studies have a reduction in the final state space and the final number of transitions. However, the reduction differs for every case study. All of the case studies have different characteristics, e.g., spare elements or cold BEs. When a DFT doesn't contain spare elements less reduction in the state space is possible, since the tool combines all activation signals.

Furthermore, the transformation of a cold BE to an I/O-IMC uses a different module template then the transformation of a warm BE to an I/O-IMC. In this module the failure while inactive is left out. Therefore, the corresponding I/O-IMC is smaller, and the reduction, which can be achieved with smart semantics, decreases compared to the reduction which can be achieved for a warm BE. However, it needs to be taken into account that application of context-dependent state space generation may influence



(a) DFT with an AND gate

(b) I/O-IMC of the DFT created by applying context-dependent state space generation

(c) I/O-IMC of the DFT created with the old implemenation of DFTCalc

Figure 11: The result in the implementation

the probability of failure for warm BEs, since their probability of failure while inactive isn't 0 and doesn't equal the probability of failure while active. Future research should analyse this problem and propose a suitable solution.

Furthermore, DFTCalc already uses some smart semantics by merging all the activation actions which happen directly when the system starts. Example 6 explains this and shows the reduction in the I/O-IMC for a DFT consisting of an AND gate with two BEs.

EXAMPLE 6. *Figure 11 shows a DFT consisting of an AND gate with two cold BEs, with a probability of failure of 0.003. Furthermore, the figure contains two I/O-IMCs, namely one while applying context-dependent state space generation and one while using the old implementation of DFTCalc. In the old implementation transitions are already merged, e.g. the activation of the three elements of the DFT happen at once. Furthermore, there is no case distinction in which of the two BEs fail. So the probability of failure in state S1 is 0.006, one of the two will fail. In the I/O-IMC created with context-dependent state space generation all activation transitions are removed. This safes one state and five transitions.*

The maximal state space during the composition of the different I/O-IMCs decreased in all of the case studies. Even though smart semantics was not applicable for all the I/O-IMCs of the elements of the DFT, the reduction is significant since the composed and minimised I/O-IMCs are smaller than the I/O-IMCs without context-dependent state space generation.

6.2 Algorithm

To determine whether a component e of a DFT is active, we constructed an algorithm, which consists of two parts, namely, procedure 1 and procedure 2. Procedure 1 implements the start of the algorithm, all elements get the state of not initialised. Afterwards we start with the initialisation, by initialising the top node of the DFT. The second part of the algorithm checks whether a component is active or not. When the component is not initialised, it is known that all of the DFT above the component is active. Therefore, the component will be initialised with an active

Table 1: State space of the case studies

Model	Tool	States	Transitions	Max States	Max transitions	P(fail)
CAS	Original	16	36	84	304	0.0460314
t=1000	Smart semantics	14	34	49	133	0.0460314
MCS	Original	18	37	6438	32202	0.9989628
t=1	Smart semantics	12	31	220	803	0.9989628
FTPP	Original	72	312	45823	230596	0.0192186
t=1000	Smart semantics	66	306	7020	32200	0.0192186
CPS	Original	39	71	918	3140	0.0013567
t=2	Smart semantics	38	70	134	291	0.0013567
SF	Original	15	36	383	1500	0.941014
t=10	Smart semantics	14	35	64	138	0.941014

Table 2: State space reduction of the case studies

Model	Original # of states	Smart # of states	Statespace reduction	Original max # of states	Smart max # of states	Maximal state space reduction
CAS	16	14	12.50 %	84	49	72.92 %
MCS	18	12	33.33 %	6438	220	96.58 %
FTPP	72	66	8.33 %	45823	7020	84.68 %
CPS	39	38	2.56 %	918	134	85.40 %
SF	15	14	6.67 %	383	64	83.29 %

state. When the component is a spare type, smart semantics can't be applied on its spare children. Therefore, all its children are initialised with being inactive. When the component itself is inactive, e.g. since it is a spare child, all of its children must be inactive as well. This part of the algorithm works recursively since it calls itself on all the children of the current component.

To apply this algorithm on a DFT with maintenance, a slight change in procedure 2 is made. In a standard DFT the primary input of a spare is activated, and when it breaks down it will stay in a failed state. However, in a DFT with maintenance, a primary input can, after a repair, become inactive. Therefore, smart semantics aren't applicable for all children of the spare. This leads to a new algorithm, namely the combination of procedure 1 and 3 in which smart semantics are not applicable for all of the children.

7. CONCLUSION

We first of all defined the different contexts to apply context-dependent state space generation in the module templates of DFTCalc. Secondly, we defined an algorithm to detect the context-type, and we generated new module templates.

7.1 Different contexts

First of all we defined two contexts in which different state space generation is desirable. The first context-type is the one in which activation of the component starts directly when the system starts. The second context type contains the elements which might be inactive for some time at the start of the system. To determine the context of a component, we defined behaviour, irrelevant behaviour and the relationship between irrelevant behaviour in an I/O-IMC and the corresponding DFT.

7.2 Module templates

We designed the module templates for the first context-type by taking the original module templates in DFTCalc, which are used for the second context type. In these module templates we deleted the activation part of the template and set the initial status to active.

7.3 Determination of context-type

To determine the context-type we implemented the algorithm as stated in 5.1. When the context-type of the component is the first, it is known that the component will start as being active. Therefore, the new module template will be applied in the conversion from DFT to I/O-IMC.

7.4 Reduction

We determined the state space reduction by applying context-dependent state space generation. For all of the five case studies there is a reduction in terms of the size of the state space, this reduction differs between 8.33 % and 33.33 %. Furthermore, the reduction of the maximal state space during composition and minimisation differs between 72.92% and 96.58%. Additionally, we calculated the reduction in computation time, which differs between 2.99% and 10.99%. These reductions in terms of the size of the state space and the computation time of the tool chain are significant.

8. FUTURE WORK

This paper proposes an algorithm for application of context-dependent state space generation in DFTCalc. However, this algorithm is only implemented for basic DFTs. In future work this algorithm could also be applied to the maintainable DFTs. Therefore, it will also be necessary to design module templates for the additional gates in maintainable DFTs, which are of context type 1.

Secondly, this paper doesn't consider the influence on the probability of failure of the context-dependent state space generation for the warm BEs. Future research should find out whether or not it is desirable to remove the activation part and the transition for failure while inactive.

9. REFERENCES

[1] F. Arnold, A. Belinfante, F. Van der Berg, D. Guck, and M. Stoelinga. Dftcalc: a tool for efficient fault tree analysis (extended version). Technical Report TR-CTIT-13-13, Centre for Telematics and

Information Technology, University of Twente, Enschede, June 2013.

[2] H. Boudali, P. Crouzen, and M. Stoelinga. A compositional semantics for dynamic fault trees in terms of interactive markov chains. In K. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, editors, *Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 441–456. Springer Berlin Heidelberg, 2007.

[3] H. Boudali, P. Crouzen, and M. Stoelinga. Dynamic fault tree analysis using input/output interactive markov chains. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 708–717, June 2007.

[4] H. Boudali, P. Crouzen, and M. Stoelinga. A rigorous, compositional, and extensible framework for dynamic fault tree analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(2):128–143, 2010.

[5] H. Boudali and J. Dugan. A discrete-time bayesian network reliability modeling and analysis framework. *Reliability Engineering and System Safety*, 87(3):337 – 349, 2005.

[6] H. Boudali and J. Dugan. A new bayesian network approach to solve dynamic fault trees. In *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, pages 451–456, Jan 2005.

[7] M. Bozzano, A. Cimatti, J. Katoen, V. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended aadl models. *The Computer Journal*, page bxq024, 2010.

[8] J. Dugan, S. Bavuso, and M. Boyd. Fault trees and sequence dependencies. In *Reliability and Maintainability Symposium, 1990. Proceedings., Annual*, pages 286–293, Jan 1990.

[9] J. Dugan, S. Bavuso, and M. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *Reliability, IEEE Transactions on*, 41(3):363–377, Sep 1992.

[10] J. Dugan, S. Bavuso, and M. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *Reliability, IEEE Transactions on*, 41(3):363–377, Sep 1992.

[11] D. Guck, J. Katoen, M. Stoelinga, T. Luiten, and J. Romijn. Smart railroad maintenance engineering with stochastic model checking. In J. Pombo, editor, *Proceedings of the Second International Conference on Railway Technology: Research, Development and Maintenance, Railways 2014, Ajaccio, Corsica, France*, volume 104 of *Civil-Comp Proceedings*, page 299, Stirlingshire, UK, April 2014. Civil-Comp Press.

[12] H. Hermanns and J. Katoen. The how and why of interactive markov chains. In F. de Boer, M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 311–337. Springer Berlin Heidelberg, 2010.

[13] S. Junges. Simplifying dynamic fault trees by graph rewriting. master thesis at rwth aachen university. 2015.

[14] 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 *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 6 pp.–, April 2006.

[15] F. Patterson-Hine and J. Dugan. Modular techniques for dynamic fault tree-analysis. In *Reliability and Maintainability Symposium, 1992. Proceedings., Annual*, pages 363–369, Jan 1992.

[16] E. Ruijters and M. Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, 15Ü16(0):29 – 62, 2015.

[17] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, JosephMinarick III, and J. Railsback. *Fault Tree Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance NASA Headquarters Washington, DC 20546, August 2002.

[18] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission Washington, D.C. 20555, January 1981.