Science of Computer Programming 196 (2020) 102478

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Synthesizing and optimizing FDIR recovery strategies from fault trees

Sascha Müller[a,*], Liana Mikaelyan[a], Andreas Gerndt[a,c], Thomas Noll[b]

[a] Software for Space Systems and Interactive Visualization, DLR (German Aerospace Center), 38108 Braunschweig, Germany
[b] Software Modeling and Verification Group, RWTH Aachen University, 52056 Aachen, Germany
[c] University of Bremen, 28334 Bremen, Germany

A R T I C L E   I N F O

Article history:
Received 3 May 2019
Received in revised form 4 May 2020
Accepted 4 May 2020
Available online 13 May 2020

Keywords:
FDIR
Fault Tree Analysis
Synthesis
Formal methods

A B S T R A C T

Redundancy concepts are major design drivers in fault-tolerant space systems. It can be a difficult task to decide when to activate which redundancy, and which component should be replaced. In this paper, we refine a methodology where recovery strategies are synthesized from a model of non-deterministic dynamic fault trees. The synthesis is performed by transforming non-deterministic dynamic fault trees into Markov automata that represent all possible choices between recovery actions. From the corresponding scheduler, optimized for maximum expected long-term reachability of failure states, a recovery strategy, optimal with respect to mean time to failure, can then be derived and represented by a model we call recovery automaton. We discuss techniques for reducing the state space of this recovery automaton, and analyze their soundness and completeness. We show that they do not generally guarantee recovery automata with the minimal number of states and derive a class where this guarantee holds. Implementation details for our approach are given and its effectiveness is verified on the basis of three case studies.

1. Introduction

Facing harsh environments with only sparse room for human intervention, space missions raise many challenges for almost any involved engineering discipline. Amongst them is the discipline of reliability engineering. Even a well-designed system still has to deal with the presence of faults. Examples for faults in the space domain may be events such as equipment failures, wrong sensor readings, random bit-flips caused by radiation, external interferences and many more. Reliability engineering aims to create fault-tolerant, stable systems that can withstand the presence of such faults to some extent and continue operating. To raise trust in handling system failures, reliability engineering tries to embed Failure Detection, Isolation and Recovery (FDIR) concepts into the system design.

Various tools and methodologies exist to derive these concepts. Fault Tree Analysis (FTA) [2] is such a methodology, commonly employed in the industry to perform state-of-the-art failure analysis [25]. A Fault Tree (FT) is a graphical failure model obtained as a result of performing an FTA. It describes how basic equipment faults propagate through the system and eventually become a system wide failure. Graphical representations of these trees are intuitive and easy to understand. FTs can provide insight into many interesting system properties. Fault combinations that lead to system failure, can be

[*] Corresponding author.
E-mail addresses: Sa.Mueller@dlr.de (S. Müller), liana.mikaelyan.18@ucl.ac.uk (L. Mikaelyan), Andreas.Gerndt@dlr.de (A. Gerndt), Noll@cs.rwth-aachen.de (T. Noll).

https://doi.org/10.1016/j.scico.2020.102478
0167-6423/© 2020 Published by Elsevier B.V.

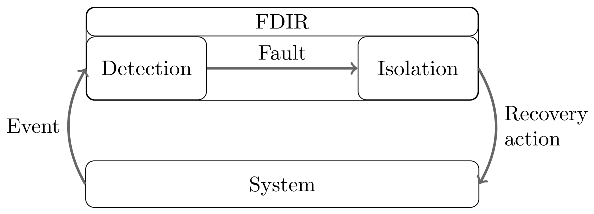


Fig. 1. FDIR and the interaction with the system.

extracted from an FT by means of qualitative analysis. Further important, computable metrics such as reliability can be obtained using quantitative analysis approaches. A popular extension of FTs are Dynamic Fault Trees (DFT). They introduce temporal dependencies and new features to analyze redundancy concepts known as spare management. However, there are challenges arising from non-deterministic behavior of DFTs. Such behavior may occur in the case of spare races. An example is a system of two operative memories together with a pool of two spare memories. If both operative memories fail at the same time, it is unclear which backup memory takes over the role of which operational one.

To overcome this shortcoming, a new methodology was presented in [21,22]. It introduces a model of Non-deterministic Dynamic Fault Trees (NdDFT) as an extension to DFTs. In contrast to the latter, the new NdDFT does not impose a fixed, rigid order on the spares to be used. As next step, the methodology foresees transforming this NdDFT model into a Markov Automaton (MA) which is suitable for the computation of the aforementioned non-deterministic decisions on spare activations. By optimizing the scheduling of the MA model in terms of system reliability, a recovery strategy for the NdDFT can be synthesized. This recovery strategy defines which spare has to be employed in which failure state of the system and can therefore guarantee an optimal reliability at all times.

The goal of the present paper is to refine the methodology initially presented in [22] by further developing an automata model that formalizes the decision process underlying a recovery strategy, a so-called Recovery Automaton (RA). We sketch its formal definition as given in [22] and show how its state space can be reduced in order to obtain an efficient implementation of recovery strategies for FDIR. Furthermore, while automata are well suited for visualization, vast state spaces in the RA can be a hindrance. A reduced automata model with a state space size graspable for a human can be visualized with human validation in mind, raising trust in the automatically generated model.

This paper is an extended version of a workshop publication [18]. It makes significant additional contributions by extending the NdDFT formalism by rate dependency (RDEP) gates, providing a refined analysis of equivalence relations between automata states, addressing the completeness of our state space reduction approach, describing a tool implementation, and presenting additional experimental results.

The remainder is structured as follows. Section 2 summarizes the related work relevant to the topic of FTs, MA, and synthesis of recovery strategies. Further background on the theory of FTs including their (non-deterministic) dynamic variants is given in Section 3. Section 4 describes the process of synthesizing recovery strategies from a given NdDFT as well as a model to represent such strategies, which is further optimized in Section 5. Section 6 gives details on the implementation and the tool-chain it has been integrated into. Section 7 then evaluates the technique on a series of use case examples. Finally, the paper concludes in Section 8 and provides some outlook to future work.

2. Related work

A failure is the inability of a system to offer certain functionality due to a fault. The goal of FDIR lies in keeping a system in a failure free state, even in the presence of faults. FDIR can generally be divided into the following main procedural steps, which may be conducted at the space as well as the ground segment of the spacecraft [28]:

- Monitor the system to detect the occurrence of faults.
- Identify the fault and localize it within the system.
- Isolate the fault and prevent propagation into further parts of the system.
- Perform recovery actions to reconfigure the system and return it into a stable state.

To perform this routine, a system is equipped with detection mechanisms that react upon system events aiming to detect faults. Based on the faults and possibly based on further diagnosis information, a recovery action is then produced. In addition, the process may also be extended by performing further bookkeeping operations such as logging. The aspect of logging, however, will not be further considered in this work. The overall relationship between FDIR, its components, and the system is visualized in Fig. 1.

Failure analysis techniques can be employed to derive the relationship between faults and how they eventually lead to a system-wide failure. One of the simplest and most popular approaches for performing failure analysis is Failure Modes and Effects Analysis (FMEA) [1]. FMEA provides a structured technique for hierarchically decomposing a system and then examining the failures in a bottom-up manner. Typically, FMEA is combined with a Criticality Analysis (FMECA), also examining severity and criticality of failures. FMEA, however, operates only on a qualitative level and does not carry enough expressive power to allow for the computation of many interesting metrics.

The technique relevant for this work is FTA. Static Fault Trees (SFT) are one of the very basic types of FTs. They employ Boolean algebra to combine different faults by AND and OR operations until they sum up to the overall system failure. The operations are also often graphically represented as gates. In contrast to FMEA, FTA is performed in a top-down fashion. The fault events are usually related to faulty components of the system. Applying this methodology, it is possible to model statements such as “The system fails if component A and component B fail” and to refine them to arbitrary levels of precision. More details on SFT are given in Section 3.1.

A popular extension to SFTs are Dynamic Fault Trees (DFT) [25]. DFTs further extend SFTs by dynamic behavior such as spare management, temporal understanding and functional dependencies. The probability of the top-level failure after time t (reliability) can be computed from a given DFT for example by transforming it into a Continuous-Time Markov Chain (CTMC) [8]. Further background on DFTs is given in Section 3.2.

Computing strategies for recovery purposes from a given fault model has been researched in other contexts. An approach similar to ours is taken in [3], which focuses on repairable fault trees. The authors consider non-deterministic repair policies where the order of repair operations is not fixed. Repairable fault trees are an extension to fault trees where faults can be transient or repaired. In their work, the repair process is realized with a new gate type called *repair boxes*. Repair boxes are equipped with a repair policy that states which resources are required for the repair process, in which order the faults should be repaired, the repair rate and so on. By then converting a repairable fault tree to a Markov decision process, an optimal repair policy (with respect to the steady state availability metric) can be computed. However, the authors do not consider DFT models.

Also focusing on repair policies and in addition on maintenance policies, [24] introduces a model of fault maintenance trees, aiming to identify optimal repair and maintenance strategies. However, instead of synthesizing these policies from the fault model, the optimal policies are obtained by testing different policies against each other by means of simulation.

Dynamic Decision Networks (DDN) are employed in [7], and their inference capabilities are exploited to create autonomous on-board FDIR systems for spacecraft that can select reactive and preventive recovery actions during run-time. The authors further consider in [23] how DDNs can be generated from an extended DFT model. Instead of off-line computing a recovery strategy with globally optimal reliability, the approach focuses on providing locally optimal (in terms of some externally provided heuristic utility functions) on-line decision making.

Timed Failure Propagation Graphs are used in [5] to synthesize FDIR components. These components are monitors for fault detection and implement recovery plans for each specified combination of fault and spacecraft mode. Here, planning-based approaches with predefined actions are employed to create the recovery components. To provide the required timing information, a developed understanding of the system implementation is often needed. Many FDIR concepts such as redundancy features, however, are ideally already finalized at this stage.

The problem of failure rates sometimes not being known a priori is tackled in [27]. The authors consider fault trees with symbolic failure rates and synthesize upper failure rate bounds for meeting reliability thresholds.

3. Fault trees

In contrast to simple fault models such as FMECA tables, fault trees offer a high degree of expressiveness. At the same time, they are simple enough to remain computationally tractable for the purpose of analysis. This makes fault trees a highly attractive modeling formalism. Syntactically, FTs are graphs consisting of two types of nodes, respectively, representing events and gates. The root node, or *top-level event (TLE)*, usually represents the occurrence of a system failure, whereas the leaves of the tree model the event of individual components failing. The leaves are also called *basic events (BE)* and can be equipped with failure rates. They correspond to Boolean variables where false represents the initial state of no failure. The variable is considered true after the occurrence of a failure event. We consider here only the case of a permanent failure, i.e., once a BE has failed, it remains in a failed state for all future points in time. The branches of the trees are represented by the gates performing operations on the events. FTs are directed acyclic graphs starting from the BEs pointing over the gates towards the system failure event. In the following, basic events will be denoted by $b_1, b_2, \ldots$, sets of basic events by $B_1, B_2, \ldots$ and failure rates by $\lambda_1, \lambda_2, \ldots$.

3.1. Static fault trees

Fig. 2 shows the gates and events used in the SFT notation. SFTs employ Boolean operations represented by AND and OR gates. Intermediate results of these operations can be given identifiable names by means of non-basic faults. Usually, the TLE is represented using a non-basic fault. There also exist other gates such as the k-VOTE gate, which propagates if at least k inputs have failed. Observe that a 1-VOTE gate corresponds to an OR gate and a k-VOTE gate with k inputs to an AND gate. Implementation-wise, all gates can therefore be considered as k-VOTE gates for some appropriate k. Some other extensions also introduce a NOT gate. However, this allows the construction of fault trees where the TLE can change from having failed to working again as new failures occur. Such fault trees are known as non-coherent fault trees and have been dismissed as being a sign for modeling errors [26].

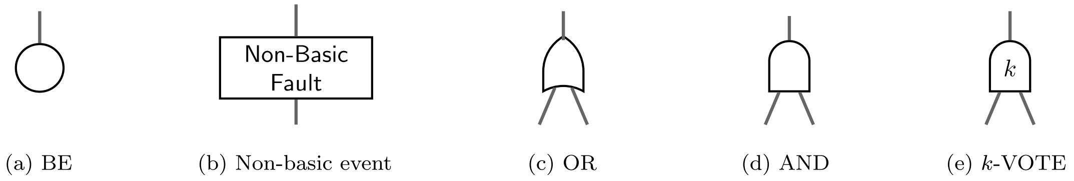


Fig. 2. Gates and events in a static fault tree.

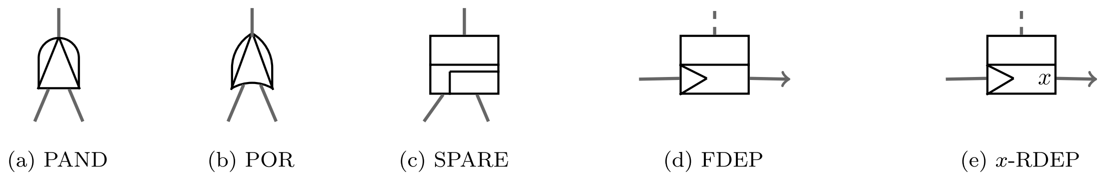


Fig. 3. Dynamic gates.

3.2. *Dynamic fault trees*

Many extensions have been proposed to the formalism of FTs [25] to increase its expressiveness and to enhance its features. One particular extension are Dynamic Fault Trees (DFT). They introduce temporal understanding and new features to analyze redundancy concepts known as spare management. In DFTs, a node can be either in a failed, active (operational) or dormant (operational) state. A node that represents an unactivated spare is dormant. All other nodes are activated. Together with this state, failure rates for failing actively and failing dormantly can be defined for every BE. Usually the dormant failure rate is lower than the active failure rate, and can possibly even be 0. By using a dormant failure rate of 0, cold redundancy can be modeled. Conversely, hot redundancy can be modeled by setting the dormant failure rate to the active failure rate.

Fig. 3 depicts the notation to extend SFTs to DFTs introducing new gates POR, PAND, SPARE, FDEP, and RDEP. The PAND (priority AND) gate propagates in case all inputs fail exactly in sequence from left to right. The POR (priority OR) gate propagates in case the leftmost input occurs before all other inputs [10]. Priority gates may come in two flavors: exclusive or inclusive. The inclusive PAND gate also propagates if the inputs occur simultaneously. On the other hand, the exclusive PAND gate only propagates if all inputs occur strictly after each other. Similarly, the exclusive POR gate only propagates if the leftmost input occurs strictly before all other inputs. It is shown in [14] that exclusive POR gates are expressive enough to model all priority gates. In this work, priority gates are considered to be exclusive.

The SPARE gate is connected to a primary event and a set of spare events. These events may be either basic or non-basic events. It propagates a failure if the primary input failed and all connected spares are either already claimed or failed themselves. The spare events can be shared with other SPARE gates. A spare can be claimed by either the one or any of the other SPARE gates. A set of spares is also referred to as a spare pool. Primary and spares may be complex fault trees themselves and even include spare gates. Hence, it is possible to nest spare gates. However, there must be no shared elements between the primary input and any spare. The order in which such a spare is chosen is deterministic and defined at design time by the reliability engineer.

The FDEP (functional dependency) has a triggering event on the left hand side and any number of functionally dependent events to the right. The former can be any event while all dependent events have to be basic. When the trigger event occurs, the dependent events are set to fail as well. Syntactically, the triggering event and the dependent events are defined to be inputs to the FDEP gate. To avoid semantic confusion, however, the graphical representation uses outgoing edges to connect dependent events. The output of an FDEP gate only indicates to which tree it belongs and has no further semantic meaning.

The x-RDEP (rate dependency) gate, based on [24], is structured similarly to the FDEP gate. When the triggering event on the left hand side occurs, the dependent events have their failure rates multiplied by the rate dependency x. The RDEP gate also has a dummy output indicating to which tree it belongs. Just like for the FDEP gate, the output gate has no further semantic meaning. Using the RDEP gate, load sharing of components can be modeled. For example, if a system is powered by two batteries and a battery fails, then the system may continue operation, but the increased load on the remaining battery might increase its likelihood to fail.

A number of additional syntactical restrictions are imposed in order to avoid semantic confusions. A fault tree is well-formed if it fulfills the following conditions:

- The fault tree has exactly one root element, the TLE.
- Spare sub-trees may not have common child nodes with other sub-trees. FDEP gates, however, may have dependent events across different sub-trees.
- FDEP gates may have any event as a trigger event, but must not induce cycles through dependent events.

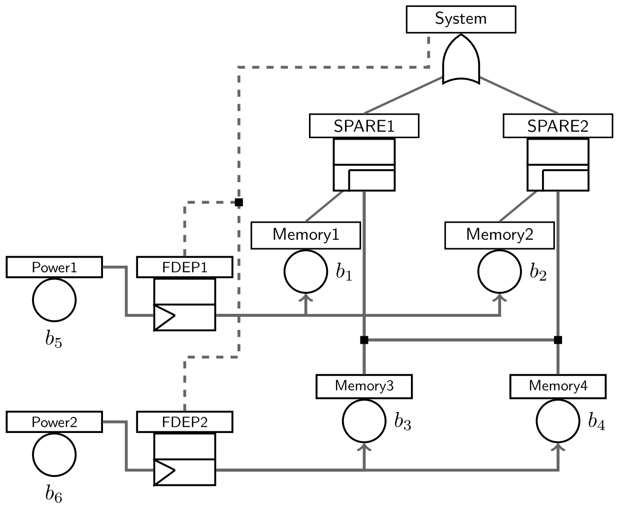


Fig. 4. Example DFT.

In the following, we give an example to illustrate the DFT notation. Fig. 4 depicts a system consisting of four memory components. Two memory components are primary components which are covered by the remaining two spare memories for failures. The two spares are part of a spare pool shared among the two SPARE gates. According to DFT semantics, priority is given to claiming Memory3 before Memory4 in case of a failure of Memory1 or Memory2. Additionally, the system is equipped with two hot redundant, always active power sources, Power1 and Power2. Power1 powers both primary components, Memory1 and Memory2. The second hot redundant source Power2 powers the redundancies, Memory3 and Memory4. Finally, FDEP gates are used to model the functional dependencies between power supplies and memory components. The FDEPs propagate the failure of a power source to the respective memory components. In the figure, dependent events of an FDEP gate are marked by an arrow. Dashed lines on the other hand indicate the dummy parent output of an FDEP gate.

3.3. Non-deterministic dynamic fault trees

As described before, DFTs require spares to be activated in a fixed and rigid order. This order cannot be adapted depending on the history of faults as they have previously occurred. This may lead to behavior such as claiming a spare from a pool despite having an already failed parent node, which might deny the spare to SPARE gates that might need it later more urgently. Additionally, in case of spare races it is not semantically clear which SPARE gate claims the actual redundancy. To relax on this semantic restriction of the DFT model, [22] introduces an inherently non-deterministic DFT model (NdDFT, following the naming in [3]). The syntax and notation of NdDFTs is completely adopted from DFTs. Semantically, the NdDFT drops the requirement that spares are always activated from left to right. Instead, the NdDFT has the choice of freely picking any spare. Moreover, the new non-deterministic semantics allows for a SPARE gate to leave the spares available for more important SPARE gates by not claiming. Whenever BEs occur in an NdDFT, the new semantics allow to perform valid recovery actions of the following form:

Definition 1 *(Recovery action).* A recovery action r in an NdDFT $\mathcal{T}$ is an action of the form

- [] (empty action) or
- CLAIM(G, S) (spare gate G claims spare S, where S is a spare of G).

We denote the set of all recovery actions possible in an NdDFT $\mathcal{T}$ by $R(\mathcal{T})$. Moreover, we extend the definition to the set of recovery action sequences by

$$RS(\mathcal{T}) := (R(\mathcal{T}) \setminus \{[]\})^*$$

For recovery action sequences, the empty action is ignored and considered as the empty word ϵ. The * here denotes the usual Kleene closure. Similarly we denote the set of all non-empty subsets of basic events of an NdDFT $\mathcal{T}$ by $BES(\mathcal{T})$.

4. Synthesizing recovery strategies

Here we describe the essential steps; details can be found in [22]. First, the NdDFT model is transformed into a Markov Automaton (MA) that represents all possible (non-deterministic) decisions on spare activations. By optimizing the scheduling

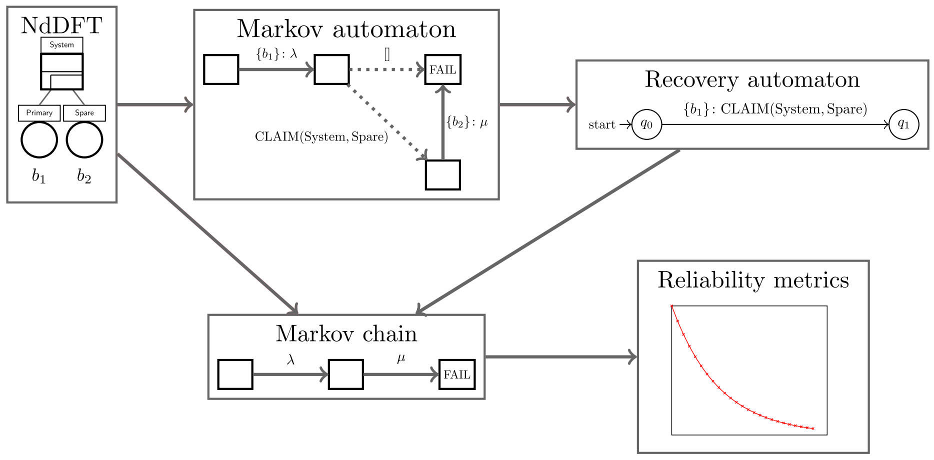


Fig. 5. Transformation road map.

of the MA model in terms of reliability of the system, a recovery strategy for the NdDFT can be synthesized. This strategy is represented by a Recovery Automaton (RA) that defines which spare has to be used in which failure state of the system and that can therefore guarantee an optimal reliability at all times. Reliability metrics can be computed by a quantitative analysis of the Markov chain that is obtained from the RA, enriched by the failure rates of basic events as determined by the original NdDFT. Fig. 5 visualizes the procedure.

The metrics computed from the Markov chain may include the optimization objective, but may also be other metrics of interest. They might in fact not only refer to the system TLE but instead employ some subsystem node as the TLE. For example, it might be of interest to query for the reliability of just a navigation subsystem. By having an explicit recovery strategy object — the recovery automaton — that is used over all subsequent metric queries, it is ensured that the same recovery behavior is always applied.

4.1. Recovery strategies and automata

To resolve the non-determinism present in NdDFTs, the actual recovery actions to be applied in failure cases are given by recovery strategies, which are implemented by recovery automata. In the following, transitions of recovery automata are labeled by a triggering set of basic events and a recovery action sequence. The elements of the triggering set are also called *guards*. Here, sets of basic events rather than single basic events are considered since FDEPs may cause several basic events to fail simultaneously. Given the observed basic events, a recovery strategy is then a mapping that returns the recovery action sequence that should be taken accordingly. The NdDFT considers recovery strategies that are composed of recovery actions as given in Definition 1. They are defined as follows:

Definition 2 *(Recovery strategy).* A recovery strategy for an NdDFT $\mathcal{T}$ is a mapping $Recovery : BES(\mathcal{T})^* \to RS(\mathcal{T})^*$ such that

- $Recovery(\varepsilon) = \varepsilon$ and
- $Recovery(B_1, \ldots, B_n) = Recovery(B_1, \ldots, B_{n-1}), rs_n$ with $rs_n \in RS(\mathcal{T})$.

As faults are permanent in the NdDFT model, each basic event can occur at most once. Hence, the recovery strategy only needs to be defined for pairwise disjoint sets of basic events, i.e., $B_i \cap B_j = \emptyset$ for $i \neq j$. Later this will ensure that each recovery strategy can be represented by a finite-state automaton that only accepts finite traces. A finite automaton that represents a recovery strategy will be called *recovery automaton*. Formally, a recovery automaton is a Mealy machine [17] having the power set of basic events $BES(\mathcal{T})$ as the input alphabet and the set of recovery action sequences $RS(\mathcal{T})$ as the output alphabet. This concept is formalized in the following:

Definition 3 *(Recovery automaton).* A Recovery Automaton (RA) $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ of an NdDFT $\mathcal{T}$ is an automaton where

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state, and
- $\delta : Q \times BES(\mathcal{T}) \to Q \times RS(\mathcal{T})$ is a deterministic transition function that maps the current state and an observed set of faults to the successor state and a recovery action sequence.

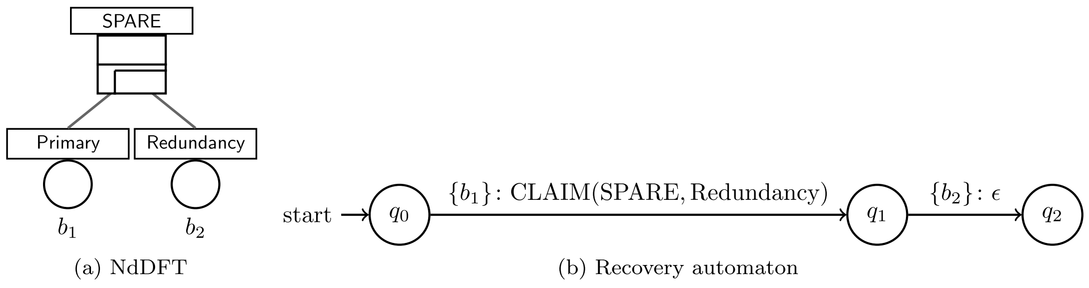


Fig. 6. Example of (a) NdDFT and (b) RA.

The transition function δ is extended to $\delta^*\colon Q \times BES(\mathcal{T})^* \to RS(\mathcal{T})^*$ by letting

$$\delta^*(q, \epsilon) := \epsilon$$
$$\delta^*(q, B \cdot w) := rs \cdot \delta^*(q', w) \text{ with } \delta(q, B) = (q', rs)$$

for any $q \in Q$, $B \in BES(\mathcal{T})$ and $w \in BES(\mathcal{T})^*$. The recovery strategy induced by a recovery automaton $\mathcal{R}$, $Recovery_{\mathcal{R}}\colon BES(\mathcal{T})^* \to RS(\mathcal{T})^*$, is given by $Recovery_{\mathcal{R}}(w) := \delta^*(q_0, w)$.

Note that in our formal definition of RA the set of states, Q, is not further specified. In Section 4.3, we will see how it can be obtained from the Markov automaton after computing an optimal scheduler. As the internal structure of those states (which basically record the history of error events occurred and the assignment of spare components) is not relevant for further optimizations, we will use symbolic states of the form q_i in our examples. In the RAs depicted in the following, if no transition is explicitly defined for some state q and some input B, then it is assumed to be an ϵ-loop transition, i.e., $\delta(q, B) = (q, \epsilon)$.

An example of a recovery automaton for a simple fault tree consisting of a SPARE gate with a cold redundant spare is given in Fig. 6. If the primary unit fails, the SPARE gate switches to the redundancy unit by claiming it.

4.2. Non-deterministic dynamic fault trees to Markov automata

Markov automata [11] are an extension of Continuous-Time Markov Chains (CTMC) [8]. They are state-based transition systems with two types of transitions: Just as CTMCs, they contain continuous-time transitions (also called *Markovian transitions*) that are labeled with rates, i.e., non-negative real values. In addition to the former, they feature immediate, non-deterministic transitions labeled by actions. Such transitions represent possible choices (for example, between recovery actions) that have to be made by a so-called *scheduler*. The computation of optimal schedulers for Markov automata with respect to various quantitative objectives, such as state reachability, is discussed in [12].

Transforming an NdDFT into a Markov automaton can be done by adapting traditional state space generation algorithms for transforming DFTs to CTMCs. As base algorithm, we use the one given in [8]. The adapted algorithm operates by memorizing two data items in each of its states: First, the history of occurred basic event sets $(B_1, B_2, \dots, B_n)$. Second, a mapping from spare gates to the currently claimed spares. The initial, empty history of the algorithm is denoted by (). Basic events which are not associated to an unactivated spare or have a dormant failure rate > 0 are considered enabled events. Starting with the initial state, all enabled events are used to compute Markovian successors for each of them while extending the history accordingly.

The respective basic event set is obtained by taking the active basic event and computing all basic events that transitively fail due to FDEPs. The transitions are labeled with the failure rate λ of the basic event causing the transition. All transitions that would lead to a state that implies that the top-level event (system failure) has occurred are connected to a special FAIL state instead. For each target state of a Markovian transition, the algorithm generates successors using non-deterministic transitions. Each non-deterministic transition is labeled by a valid recovery action.

An example of the construction is given in Fig. 7, transforming the NdDFT from Fig. 6a to a Markov automaton. After $\{b_1\}$ fails, there are two possible recovery actions: the empty action ([]) or the activation of the spare redundancy (Claim(SPARE, Redundancy)). The active failure rates of the basic events are assumed to be 1 for b_1 and b_2. The dormant failure rates are 0 for both events. Note that b_2 is initially dormant. Hence, the initial state () has only one successor, which is reachable by a transition labeled with $\{b_1\}\colon 1$. Dotted edges represent the non-deterministic transitions and solid lines represent the Markovian transitions. $B\colon \lambda$ denotes that the basic event set B occurs (actively or dormant) with rate λ. In this simple example, it is obvious that immediately activating the redundancy upon observing $\{b_1\}$ is the correct course of action. A recovery strategy that could be synthesized in this example would thus yield Recovery($\{b_1\}$) = Claim(SPARE, Redundancy).

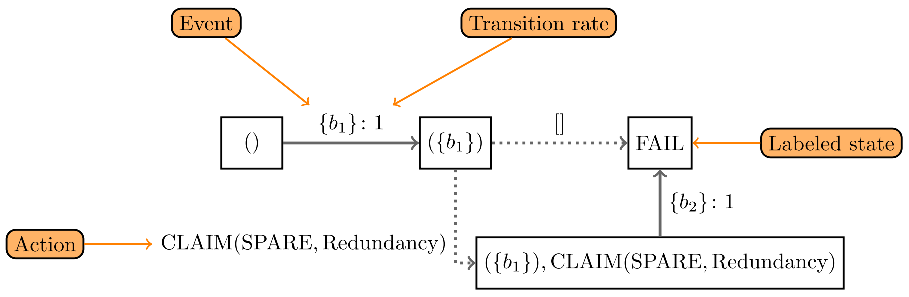


Fig. 7. Example transformation of NdDFT to MA.

4.3. Synthesizing recovery automata from Markov automata

Using existing techniques for optimizing the scheduling of a Markov automaton, the choice of non-deterministic transitions that maximizes system reliability can be computed. The recovery automata model is then used to represent the underlying decision process of the scheduler. System reliability is commonly measured by one of the following metrics:

- **Reliability After Time** t describes the probability that a system is still functional after a time span t.
- **Mean Time To Failure** (MTTF) describes the expected time span that will pass until the system-level failure occurs.
- **Fault Tolerance** describes the minimum number of faults that must occur to cause a system failure.

While the concrete metric itself is interchangeable, in this work, we focus on optimizing with regard to quantitative metrics. In particular, we concentrate on the MTTF. This gives the advantage of dropping the time parameter t required by the "reliability after time t" metric. For the Markov automaton, maximizing the MTTF corresponds to maximizing the expected long-term reachability property of the FAIL state.

Extracting a recovery automaton from a scheduler for a Markov automaton is achieved by replacing sequences of transitions for states $s_0, s_1, \dots, s_n$ of the form $s_0 \xrightarrow{B:\lambda} s_1 \xrightarrow{r_1} s_2 \xrightarrow{r_2} \dots s_{n-1} \xrightarrow{r_{n-1}} s_n$, where B is a basic event set, λ a failure rate and $r_1, \dots, r_n$ are recovery actions, with the transition $\delta(s_0, B) = (s_n, r_1 \dots r_n)$ where empty recovery actions are ignored. Observe that multiple recovery actions from the Markov automaton are combined into one recovery action sequence in the recovery automaton. This applies to all transitions where $s_1, \dots, s_n$ are the successors computed by the optimized schedule of the Markov automaton. If a state s in the MA does not define a transition for some input B, then in the recovery automaton the transition $\delta(s, B) = (s, \epsilon)$ is inserted. All other non-deterministic transitions are then discarded. Finally, the algorithm discards all unreachable states.

5. Further optimization of recovery automata

Complex systems usually exhibit a large number of faults that may occur. This means that NdDFTs describing such systems may be very large, and thus synthesized recovery automata may contain redundant states. In this section, we refine the given synthesis procedure by discussing some techniques for reducing the state space and the transition count of a synthesized recovery automaton. This leads to the task of finding an automaton with the same "behavior" that exhibits a smaller number of states. To capture this notion of having the same behavior, we introduce the concept of recovery equivalence between recovery automata as follows:

Definition 4 *(RA recovery equivalence).* Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ and $\mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ be two RAs. We define a binary relation $\sim_R$ such that it holds true that $\mathcal{R}_1 \sim_R \mathcal{R}_2$ iff for any sequence of sets of basic events $B_1, \dots, B_n$ with $B_i \cap B_j = \emptyset$ for any $i \neq j$ it holds that:

$$Recovery_{\mathcal{R}_1}(B_1, \dots, B_n) = Recovery_{\mathcal{R}_2}(B_1, \dots, B_n).$$

Given a recovery automaton as an input, the task of minimization means to compute an equivalent recovery automaton with as few states as possible. The standard problem of automata minimization is well-known and has been studied extensively. In this work, we apply the usual definition of trace equivalence and lift it to states of recovery automata:

Definition 5 *(Trace equivalence).* Let $\mathcal{R}_\mathcal{T} = (Q, \delta, q_0)$ be an RA. A trace equivalence $\approx \subseteq Q \times Q$ is a maximal binary relation such that it holds for any states $q_1, q_2 \in Q$ that $q_1 \approx q_2$ iff for any $B \in BES(\mathcal{T})$ it holds that:

$$\delta(q_1, B) = (q_1', rs_1) \text{ and } \delta(q_2, B) = (q_2', rs_2) \text{ with } q_1' \approx q_2' \text{ and } rs_1 = rs_2$$

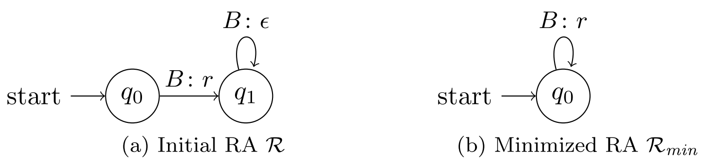

start
q_0
$B: r$
q_1
$B: \epsilon$
(a) Initial RA $\mathcal{R}$
start
q_0
$B: r$
(b) Minimized RA $\mathcal{R}_{min}$

Fig. 8. Example for merging non-trace-equivalent states.

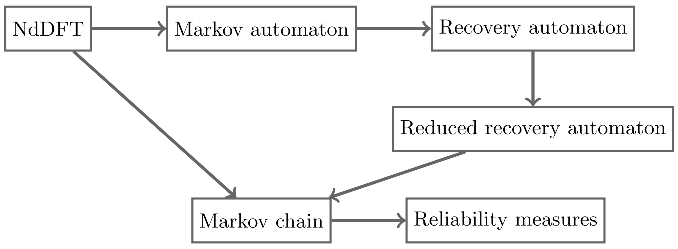


Fig. 9. Updated transformation workflow.

Equivalent states in an automaton can be computed using a partition refinement algorithm [13]. It operates by iteratively partitioning states so that states in different partitions are guaranteed to be inequivalent. On automata with acceptance conditions, this is usually achieved by using final and non-final states as the initial partitions. In this setting, the initial partitions are created using matching input-output mappings. The algorithm then proceeds to refine the partitions by identifying sub-partitions with different output behavior. When the algorithm terminates, each partition contains equivalent states only. A minimized automaton can then be obtained by merging all equivalent states. In the setting of recovery automata, we can go even further and merge pairs of states that are not trace-equivalent as long as the behavior of the automaton does not change. A simple example for a case where merging non-equivalent states yields a recovery automaton that induces an equivalent recovery strategy can be seen in Fig. 8.

In the example, the states q_0 and q_1 are clearly not trace-equivalent since q_0 outputs r upon reading B whereas q_1 outputs ϵ upon reading B. However, it can be shown that the two automata are recovery-equivalent: Since B is the only input, the traces fulfilling the condition of having no input repetition according to Definition 4, i.e., $B_1, \dots, B_n$ with $B_i \cap B_j = \emptyset$ for any $i \neq j$, are exactly B and ϵ. Furthermore, it holds that $Recovery_{\mathcal{R}}(\epsilon) = \epsilon = Recovery_{\mathcal{R}_{min}}(\epsilon)$ and $Recovery_{\mathcal{R}}(B) = r = Recovery_{\mathcal{R}_{min}}(B)$ by Definition 3. The equivalence of the two recovery automata then follows by Definition 4. Intuitively, the transition $B: \epsilon$ in the initial RA can never be taken due to B being disabled, as it must have already occurred. In the following, we present the main contribution of this work: Rules that allow to optimize the state space beyond merging trace-equivalent states, yet yield implementations of equivalent recovery strategies. We identify two cases where we can optimize the state space without affecting the induced recovery strategy.

- **Case 1: Optimizing Orthogonal States.**
- **Case 2: Optimizing the FAIL state.**

In both cases, the key to minimization that we exploit is the fact that the inputs of the automaton are produced by an FT. Hence, basic events can only occur at most once. This leads to the effect that certain traces in the RA are invalid inputs for the induced recovery strategy. Therefore, this restriction gives us additional freedom for merging states that would not be allowed to be merged in a standard automata model. The updated workflow with the integrated state space reduction for the recovery automaton is given in Fig. 9.

5.1. Optimizing orthogonal states

In the first rule, the idea is to identify states that may have transitions with disagreeing outputs but where we can guarantee for certain that conflicts are excluded, as their necessary inputs can no longer be produced. As mentioned before, the key to this idea lies in the exploitation of the property that basic events can only occur at most once in an FT. This leads to the following observation: If a basic event occurs on every path leading to a certain state in an RA, then it is guaranteed that in the future no transition listing this basic event in its guards can be taken. Note that recovery automata are deterministic automata, meaning that unlike non-deterministic automata, they always have a transition defined for every possible input. Fig. 10 abstractly illustrates the application of this merging rule. In order to reach q_1, the event set B_2 must occur. Therefore, upon reaching q_1, the transition $B_2 : z_2$ can no longer be taken. Similarly, the transition $B_1 : z_1$ can longer be taken once q_2 is visited. Hence, the states q_1 and q_2 can be safely merged, without changing the recovery-equivalence of the automaton, by eliminating the disabled transitions.

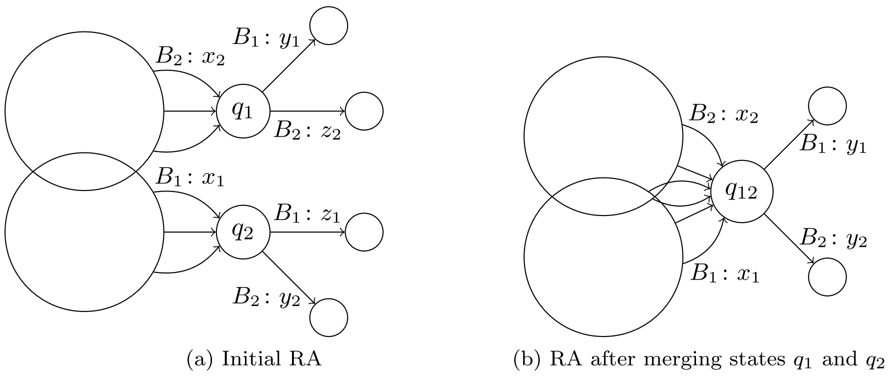


Fig. 10. Merging orthogonal states.

For the purpose of formalizing the intuitively given notion, we now introduce the concept of *orthogonal states*. To capture the basic event sets that can no longer be produced by an FT upon having reached a state in the RA, we define the set of *disabled inputs* of a state q as a function $DI\colon Q \to 2^{BES(\mathcal{T})}$ with:

$$DI(q) := \{B \in BES(\mathcal{T}) \mid \text{ for all paths } q_0 \xrightarrow{B_0 : rs_0} q_1 \xrightarrow{B_1 : rs_1} \ldots q_{n-1} \xrightarrow{B_{n-1} : rs_{n-1}} q \ \exists i : B_i \cap B \neq \emptyset\}.$$

Notice that the intersection operation in this definition suffices, since if along every path to a state q at least one basic event $b \in B$ happens, then the event set B as a whole cannot happen after visiting q. In order to compute the set of disabled inputs, we perform a data flow analysis. For this we apply the work list algorithm [15], which operates by propagating data flow information along the edges of a graph structure. For each node, the data flow information is combined using a *transfer function*. For computing the disabled inputs, the following transfer functions are employed:

$$DI(q_0) := \emptyset$$
$$DI(q) := \bigcap_{(p,B)\in pred(q)} DI(p) \cup \{B\}$$

with $pred(q) := \{(p, B) \mid \delta(p, B) = (q, rs) \text{ for some } rs, p \neq q\}$ denoting the set of predecessor transitions of a state q. Having setup these preliminary definitions, the concept of orthogonality between states can now be formalized with the following definition:

Definition 6 *(Orthogonal states).* Let $\mathcal{R}_{\mathcal{T}} = (Q, \delta, q_0)$ be an RA. Let further $p, q \in Q$ be two non-initial distinct states and $B \in BES(\mathcal{T})$. Then p, q are orthogonal with respect to B iff

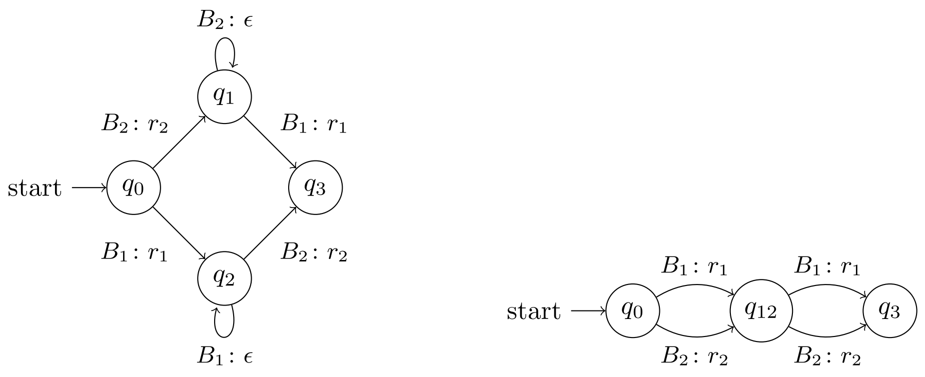
$$B \in DI(p) \cup DI(q).$$

If upon reaching a state p it is guaranteed that an event in B has occurred, then it is guaranteed that B will not happen starting from state p. Therefore, it does not matter which recovery action will be defined for B in state p, since the recovery action will never be applied. Thus, if in state q a specific recovery action has to be performed for B, then we can choose the same recovery action for B in the combined state. Informally, if this holds for all events B, then we can think of states p and q as equivalent, since their behavior in terms of recovery actions is the same, and therefore such states can be merged.

To illustrate the optimization rule based on the definition of orthogonality, we consider as an example the recovery automaton depicted in Fig. 11. This RA reacts to two distinct basic event sets B_1 and B_2 and performs a corresponding recovery action r_1 or r_2 accordingly. An NdDFT that would produce such an RA would be for example a system consisting of two parallel spare gates running independently from each other, e.g., spare gates with no shared spare. For the disabled inputs we have:

- $DI(q_0) = \emptyset$,
- $DI(q_1) = DI(q_0) \cup \{B_2\} = \{B_2\}$,
- $DI(q_2) = DI(q_0) \cup \{B_1\} = \{B_1\}$ and
- $DI(q_3) = (DI(q_1) \cup \{B_1\}) \cap (DI(q_2) \cup \{B_2\}) = \{B_1, B_2\}$.

Thus, by Definition 6 it holds that q_1 and q_2 are orthogonal with respect to basic event sets B_1 and B_2. Observe that q_1 has an outgoing loop labeled with $B_2 : \epsilon$ that is disabled. Similarly, q_2 has an outgoing loop labeled by $B_1 : \epsilon$ that cannot occur. In the merged RA, these transitions are eliminated, and all the other incoming and outgoing transitions are redirected to start and end at the merged state, respectively.



(a) Initial RA (b) RA $\mathcal{R}_2$ after merging states q_1 and q_2

Fig. 11. Example merge of orthogonal states.

We are now ready to incorporate the concept of orthogonality into an equivalence definition for states of recovery automata. For this purpose, we enrich the basic trace equivalence definition from Definition 5 as follows:

Definition 7 *(Syntactical state recovery equivalence).* Let $\mathcal{R}_\mathcal{T} = (Q, \delta, q_0)$ be an RA. A state-based recovery equivalence $\approx_R \subseteq Q \times Q$ is a maximal relation such that it holds for any states $q_1, q_2 \in Q$ that $q_1 \approx_R q_2$ iff for any $B \in BES(\mathcal{T})$ it holds that either:

- $\delta(q_1, B) = (q'_1, rs_1)$ and $\delta(q_2, B) = (q'_2, rs_2)$ with $q'_1 \approx_R q'_2$ and $rs_1 = rs_2$ or
- q_1, q_2 are orthogonal with respect to B.

Note that all conditions are syntactical, and we have introduced the necessary means to check them. In the next step, we aim to formalize the merging procedure for obtaining a new recovery automaton given two recovery-equivalent states. For this we define an *orthogonal merge* operation. Following the process described in the previous example yields the following formal definition, which is visualized by Fig. 10:

Definition 8 *(Orthogonal merge).* Let $\mathcal{R}_\mathcal{T} = (Q, \delta, q_0)$ be an RA. The orthogonal merge of recovery-equivalent states $q_1, q_2 \in Q$ is defined by the function $OM(\mathcal{R}, q_1, q_2) := (Q', \delta', q'_0)$ with:

- $Q' := Q \setminus \{q_1, q_2\} \uplus \{q_{12}\}$,
- for all $B \in BES(\mathcal{T})$,
  - $\delta'(q_{12}, B) := \delta(q_2, B)$ if $B \in DI(q_1)$ and
  - $\delta'(q_{12}, B) := \delta(q_1, B)$ if $B \in DI(q_2)$,
- $\delta'(p, B) = (q_{12}, rs)$ for any $B \in BES(\mathcal{T})$ and p such that $\delta(p, B) = (q_1, rs)$ or $\delta(p, B) = (q_2, rs)$ and
- $q'_0 := q_0$ if $q_1 \neq q_0$, and $q_2 \neq q_0$ and $q'_0 = q_{12}$ otherwise.

Intuitively, the operation replaces the states to be merged by a new combined state, adjusting the transition function accordingly. The initial state is updated in case one of the states is also the initial state. Transitions with enabled guards are copied (where the result is guaranteed to be unique). On the other hand, transitions with disabled guards are discarded. We now prove the soundness of the orthogonal merge. The following theorem states under which conditions merging two recovery-equivalent states yields a recovery-equivalent recovery automaton.

Theorem 1. *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ be an RA and $q_1, q_2 \in Q$ such that $q_1 \approx_R q_2$ and for any $B \notin DI(q_1) \cup DI(q_2)$, it holds for all q'_1, q'_2 such that $\delta(q_1, B) = (q'_1, rs)$ and $\delta(q_2, B) = (q'_2, rs)$, we have $q'_1 \approx_R q'_2$ implies $q'_1 = q'_2$. That is, all equivalent successors have already been merged. Let further $OM(\mathcal{R}_1, q_1, q_2) = \mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ be an RA resulting from the orthogonal merge of q_1 and q_2. Then $\mathcal{R}_1 \approx_R \mathcal{R}_2$.*

Proof. Let $\beta := B_1, \ldots, B_n \in BES(\mathcal{T})^*$ be a sequence of basic event sets produced by an NdDFT. Then $B_i \cap B_j = \emptyset$ for any $i \neq j$. We distinguish two cases:

- Assume $\mathcal{R}_1$ never visits q_1 or q_2. By definition of $\mathcal{R}_2$ we then have that also $\mathcal{R}_2$ does not visit q_{12}. And by definition of $\mathcal{R}_2$ again we thus immediately have that $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.
- Assume $\mathcal{R}_1$ visits q_1 (the case of visiting q_2 is analogous) upon reading B_i for some $i < n$. Now consider B_{i+1}. Let q'_1, q'_{12} and rs_1, rs_{12} be such that:

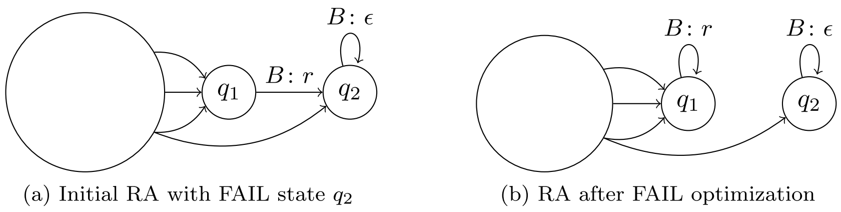

$B:\epsilon$
q_1
$B:r$
q_2
(a) Initial RA with FAIL state q_2
$B:r$
$B:\epsilon$
q_1
q_2
(b) RA after FAIL optimization

Fig. 12. Application of FAIL optimization.

$$\delta_1(q_1, B_{i+1}) = (q'_1, rs_1) \text{ and}$$
$$\delta_2(q_{12}, B_{i+1}) = (q'_{12}, rs_{12}).$$

By Definition 7 this means that we have either:

- q_1, q_2 are orthogonal with respect to B_{i+1}. Then by Definition 6 it holds that:

  $$B_{i+1} \in DI(q_1) \cup DI(q_2)$$

  Assume $B_{i+1} \in DI(q_1)$. Then there exists by construction of DI an index $j < i+1$ such that $B_{i+1} \cap B_j \neq \emptyset$. Contradiction to the definition of β. Hence $B_{i+1} \notin DI(q_1)$. But since $B_{i+1} \in DI(q_1) \cup DI(q_2)$, this implies $B_{i+1} \in DI(q_2)$. Note that $DI(q_2) \subsetneq DI(q_1)$ or otherwise we obtain again a contradiction to the construction of β. By construction of $\mathcal{R}_2$ and Definition 8 this implies that:

  $$(q'_{12}, rs_{12}) = \delta_2(q_{12}, B_{i+1}) = \delta_1(q_1, B_{i+1}) = (q'_1, rs_1)$$

  Hence we can conclude $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.
- q_1, q_2 are not orthogonal with respect to B_{i+1}. Then $B_{i+1} \notin DI(q_1) \cup DI(q_2)$ and $rs_1 = rs_{12}$ and $q'_1 \approx_R q'_{12}$. But then by assumption we get $q'_1 = q'_{12}$. We hence obtain $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.

In all cases we have $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$ and thus $\mathcal{R}_1 \approx_R \mathcal{R}_2$ by Definition 4. □

5.2. Optimizing the FAIL state

The idea of the second case is to identify FAIL states that do not contribute to new recovery action sequences when a set of faults occurs. If a state only leads to a FAIL state, the transition can be turned into a self-loop. And should the FAIL state no longer be reachable, it can be eliminated. This rule is abstractly illustrated in Fig. 12. We further introduce the concept of a FAIL state for recovery automata.

Definition 9 *(FAIL state).* Let $\mathcal{R}_\mathcal{T} = (Q, \delta, q_0)$ be an RA and $q \in Q$ a state. Then q is a FAIL state iff for any $B \in BES(\mathcal{T})$, all transitions from q are of the form $\delta(q, B) = (q, \epsilon)$.

A FAIL state is thus simply a sink state with only ϵ-loop transitions. With the definition of the FAIL state we can now also formally define an optimization operation similar to the orthogonal merge operation.

Definition 10 *(FAIL optimization).* Let $\mathcal{R}_\mathcal{T} = (Q, \delta, q_0)$ be an RA. The FAIL optimization of a state $q_1 \in Q$ with respect to a FAIL state $q_2 \in Q$ is given by the function $FO(\mathcal{R}, q_1, q_2) := (Q', \delta', q_0)$ with:

- $Q' := Q \setminus \{q_2 \mid \neg\exists p \in Q \setminus \{q_2\}, B \in BES(\mathcal{T}) : \delta(p, B) = (q_2, rs)\}$ and
- $\delta'(q_1, B) := (q_1, rs)$ if $\delta(q_1, B) = (q_2, rs)$ for all $B \in BES(\mathcal{T})$.

Note that the FAIL optimization is not applicable to all pairs of states and FAIL states. We show this property in the following lemma. Intuitively, when a state has multiple non-ϵ transitions, then the FAIL rule cannot be applied.

Lemma 1. *There exists a recovery automaton $\mathcal{R} = (Q, \delta, q_0)$ with state $p \in Q$ and FAIL state $q \in Q$ such that $\mathcal{R}$ is not recovery-equivalent to $FO(\mathcal{R}, p, q)$.*

Proof. Consider the recovery automaton $\mathcal{R} := (Q, \delta, q_0)$ and states q_0, q_1 as given in Fig. 13a. We show that $\mathcal{R}$ and the states q_0, q_1 fulfill the conditions. Let further $\mathcal{R}' := FO(\mathcal{R}, q_0, q_1)$ be the recovery automaton obtained through the FAIL optimization of q_0 with respect to the FAIL state q_1, as depicted in 13b. For the input sequence $\beta := B_1 B_2$ we now have $Recovery_\mathcal{R}(\beta) = r_1 \neq r_1 r_2 = Recovery_{\mathcal{R}'}(\beta)$. By Definition 7 we thus obtain that $\mathcal{R}$ and $\mathcal{R}'$ are not recovery-equivalent. □

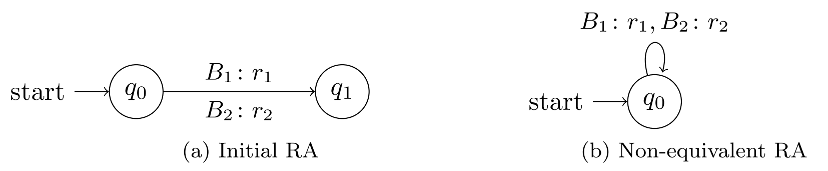


Fig. 13. RA where the FAIL rule is not applicable.

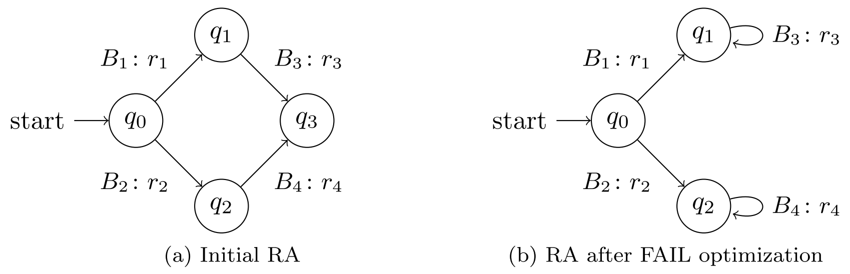


Fig. 14. RA that requires FAIL rule for optimization.

Next, we show that the FAIL optimization rule is not covered by the orthogonal merge rule. This is done by giving a concrete example where the orthogonal merge cannot remove any states, yet through application of the FAIL rule, a smaller recovery automaton can be obtained.

Lemma 2. *There exists a recovery automaton $\mathcal{R} = (Q, \delta, q_0)$ such that:*

- *for any states $p, q \in Q$ with $p \neq q$ it holds that $p \not\approx_R q$ and*
- *there exists a recovery-equivalent recovery automaton with fewer states that can be obtained through the FAIL optimization rule.*

Proof. Consider the following recovery automaton $\mathcal{R} := (Q, \delta, q_0)$ as illustrated in Fig. 14a. We show in the following that $\mathcal{R}$ fulfills the desired conditions. First of all, it holds that:

$$B_3, B_4 \notin DI(q_1) \cup DI(q_2) = \{B_1\} \cup \{B_2\}$$

Furthermore, for any states $p, q \in Q$ with $p \neq q$, there exists a B such that $\delta(q, B) = (r, q'), \delta(p, B) = (r', p')$ with $r \neq r'$. Taking these two together, we directly obtain $p \not\approx_R q$ for any states $p \neq q \in Q$ by Definition 7. On the other hand, q_3 is a FAIL state. Also both pairs q_1, q_3 and q_2, q_3 fulfill the conditions for sound application of the FAIL rule according to Theorem 2, as they possess exactly one non-ϵ transition leading into a FAIL state. The reduced RA $\mathcal{R}' := FO(FO(\mathcal{R}, q_1, q_3), q_2, q_3)$ with the FAIL state eliminated is illustrated in Fig. 14b. □

To conclude on the FAIL rule, we prove the soundness of the FAIL optimization. The conditions for soundness of the formalized optimization operation can be captured by the following theorem:

Theorem 2. *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ be an RA with a pair of states q_1 and q_2 such that q_2 is a FAIL state and all transitions of q_1 are ϵ-loops except for one transition being of the form $\delta_1(q_1, B) = (q_2, rs)$, such that $rs \neq \epsilon$. Let further $FO(\mathcal{R}, q_1, q_2) = \mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ be the FAIL-optimized RA. Then $\mathcal{R}_1 \approx_R \mathcal{R}_2$.*

Proof. Let $\beta := B_1, \ldots, B_n \in BES(\mathcal{T})^*$ be a sequence of basic event sets produced by an NdDFT. Then $B_i \cap B_j = \emptyset$ for any $i \neq j$. We distinguish two cases:

- Assume $\mathcal{R}_1$ never visits q_1. Then by definition of $\mathcal{R}_2$, it also never visits q_1. As both automata are defined to be equal otherwise, we then immediately have that $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$.
- Assume $\mathcal{R}_1$ visits q_1 upon reading B_i for some $i < n$. Then by definition, $\mathcal{R}_2$ also visits q_1 upon reading B_i. Now consider B_{i+1}. By the construction of $\mathcal{R}_2$, it holds that $\delta_1(q_1, B_{i+1}) = (q_2, rs)$ and $\delta_2(q_1, B_{i+1}) = (q_1, rs)$ for some recovery action sequence rs (I). Since q_2 is a FAIL state, we obtain from Definition 9 that $\delta_1(q_2, B_j) = (q_2, \epsilon)$ for any $j > i + 1$ (II). Moreover, since also $B_j \cap B_{i+1} = \emptyset$ for any $j > i + 1$ we also have by definition of q_1 and $\mathcal{R}_2$ that $\delta_2(q_1, B_j) = (q_1, \epsilon)$ (III). In total, we can therefore conclude that:

$$Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_1}(B_1, \ldots, B_n) \qquad \text{(Def. } \beta\text{)}$$

$$\begin{aligned}
&= Recovery_{\mathcal{R}_1}(B_1, \dots, B_i, B_{i+1}) && \text{(II)}\\
&= Recovery_{\mathcal{R}_2}(B_1, \dots, B_i, B_{i+1}) && \text{(I)}\\
&= Recovery_{\mathcal{R}_2}(B_1, \dots, B_n) && \text{(III)}\\
&= Recovery_{\mathcal{R}_2}(\beta) && (\text{Def. } \beta)
\end{aligned}$$

In all cases $Recovery_{\mathcal{R}_1}(\beta) = Recovery_{\mathcal{R}_2}(\beta)$. Hence, $\mathcal{R}_1 \approx_R \mathcal{R}_2$ by Definition 4. □

5.3. Completeness

Having established two optimization rules, the question arises if these rules are now complete. That is, if we consider an arbitrary recovery automaton, do the optimization rules yield a minimal recovery automaton at all times? This question is investigated in this section. The investigation is carried out roughly according to the following steps: First, we define the recovery equivalence relation on the state level to allow for comparison with the state-based syntactical recovery equivalence. Using this, we show that the optimization rules we provided are indeed generally not sufficient to characterize recovery equivalence. In the following, we then define a class of recovery automata that excludes the problematic instances and for which we can thus prove the completeness of our approach.

Definition 11 *(Semantic state recovery equivalence).* Semantic state-based recovery equivalence $\sim_R \subseteq Q \times Q$ is a maximal relation such that it holds for any states $q_1, q_2 \in Q$ that $q_1 \sim_R q_2$ iff $\delta^*(q_1, w) = \delta^*(q_2, w)$ for any $w = B_1, \dots, B_n$ with

- $B_i \cap B_j = \emptyset$ for any $i \neq j$ and
- $B_i \notin DI(q_1) \cup DI(q_2)$ for all i.

The definitions of state-based and automata-based recovery equivalence are nearly identical, except that the former disregards basic event sets that are excluded by disabled inputs. This additional property ensures that we can only continue from a given state using basic event sets that are not disabled due to having already occurred. The connection between both concepts is given through the recovery equivalence of the initial states. This property is captured by the following lemma.

Lemma 3. *Let $\mathcal{R}_1 = (Q_1, \delta_1, q_{01})$ and $\mathcal{R}_2 = (Q_2, \delta_2, q_{02})$ then $\mathcal{R}_1 \approx_R \mathcal{R}_2$ iff $q_{01} \sim_R q_{02}$.*

Proof. We have $DI(q_{01}) = DI(q_{02}) = \emptyset$ due to q_{01}, q_{02} being initial states. Furthermore, it holds that:

- $\delta^*(q_{01}, B_1 \dots B_n) = Recovery_{\mathcal{R}_1}(B_1, \dots, B_n)$ and
- $\delta^*(q_{02}, B_1 \dots B_n) = Recovery_{\mathcal{R}_2}(B_1, \dots, B_n)$,

for any sequence $B_1, \dots, B_n$ by definition of δ^*. Hence we obtain the equivalence by Definition 11 and Definition 4. □

5.3.1. General incompleteness

We now show that our approach is generally not optimal. This is achieved by constructing a recovery automaton that cannot be optimized further by any of the provided optimization rules but still contains recovery-equivalent states. We also show that exploiting these recovery-equivalent states, it is indeed possible to obtain a smaller recovery-equivalent recovery automaton.

Lemma 4. *There exists a recovery automaton $\mathcal{R}$ such that:*

- $\sim_R \nsubseteq \approx_R$,
- *the FAIL rule cannot be applied and*
- *there exists a recovery-equivalent recovery automaton with fewer states.*

Proof. Consider the recovery automaton $\mathcal{R} := (Q, \delta, q_0)$ as illustrated in Fig. 15a. We show in the following that $\mathcal{R}$ fulfills all listed conditions. Consider further the states q_1 and q_2. We have the following two properties:

- Both q_1 and q_2 yield the recovery action r_0 upon reading B and ϵ for any other input.
- In the successors p_1 and p_2, the input B can no longer be read if reached via q_1 or q_2, respectively.

Hence $q_1 \sim_R q_2$ by Definition 11. Furthermore, since p_1 and p_2 yield $r_1 \neq r_2$ respectively upon reading B, it also holds that $p_1 \nsim_R p_2$. We also have that $DI(p_1) = \emptyset$ and $DI(p_2) = \emptyset$ and thus in particular $B \notin DI(p_1) \cup DI(p_2)$. Hence p_1 and p_2 are not orthogonal with respect to B. In total we also obtain $p_1 \not\approx_R p_2$ by Definition 6. Furthermore, we also have

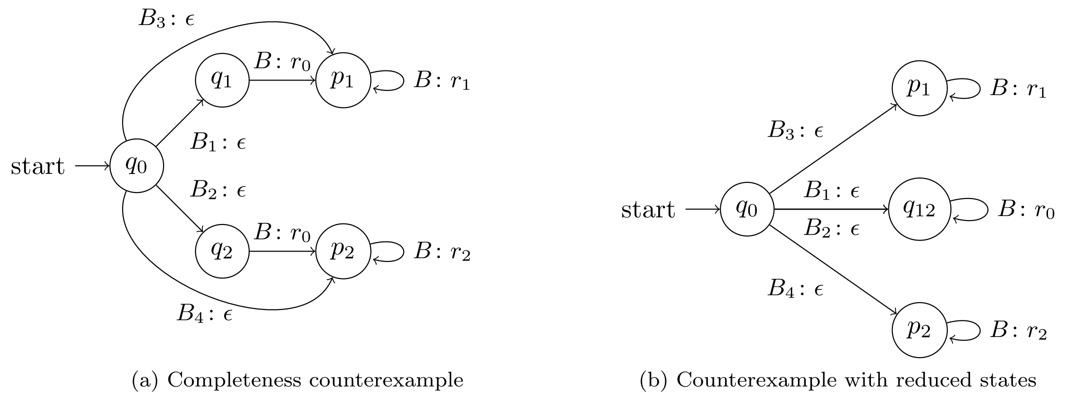


Fig. 15. Completeness counterexample that cannot be obtained with optimization rules.

$B \notin DI(q_1) \cup DI(q_2) = \{B_1\} \cup \{B_2\}$. Thus again by Definition 6, we finally obtain $q_1 \not\approx_R q_2$, which implies $\sim_R \not\subseteq \approx_R$. As for the FAIL rule, clearly there exists no FAIL state, therefore, it cannot be applied. Fig. 15b shows the optimized counterexample RA with fewer states. □

5.3.2. Restricted completeness

The key issue leading to incompleteness using a merging-based approach lies in the property that it is generally possible for two states to be recovery-equivalent but to have non-recovery-equivalent successors. This property was also present in the counterexample of Lemma 4. Requiring equivalent successors, however, is an intrinsic part of the definition of $\approx_R$. The only exception are successors that cannot be reached due to the respective guards being in the set of disabled inputs. We show in the following that if we focus on the class of recovery automata that exclude this property, we can obtain the desired completeness property.

Definition 12 *(Successor consistency).* An RA is consistent with respect to $\sim_R$ iff for any p, q with $p \sim_R q$ it holds for any $B \notin DI(p) \cup DI(q)$ that $p' \sim_R q'$. The class of RAs consistent with respect to $\sim_R$ is defined as

$$\mathcal{C} := \{\mathcal{R} \mid \mathcal{R} \text{ is consistent with respect to } \sim_R\}$$

In addition to being a member of the class $\mathcal{C}$, to simplify the proofs, we also introduce a normal form for recovery automata in the following. This normal form will guarantee later on that it is possible to reach each state backwards from a sink state.

Definition 13 *(Normal form).* A recovery automaton $\mathcal{R}$ is in normal form iff there exist no cycles, except for loop-transitions.

A recovery automaton can be easily transformed into normal form. The idea is to again exploit the property that every basic event set can occur at most once.

Lemma 5. *Each RA $\mathcal{R}$ can be transformed into a recovery-equivalent RA in normal form.*

Proof. Since every input can occur at most once, an equivalent RA can be gained by loop unrolling and by removing disabled transitions. □

However, unrolling an RA may lead to the introduction of new states. Since the goal is the reduction of states, this is undesirable. In the special case of recovery automata synthesized from an FT, according to the given construction, however, no additional states need to be introduced. This is due to the fact that these recovery automata are already in normal form.

Lemma 6. *If $\mathcal{R}$ is an RA synthesized from an FT, then $\mathcal{R}$ is already in normal form.*

Proof. Since faults are permanent, the given state space generation for transforming an NdDFT to a MA creates successor states with strictly monotonically increasing failed events. Hence, the algorithm does not add any cycles. The extraction process of an RA from an MA chiefly involves removing nodes and edges, and only introduces new edges of the form $\delta(q, B) = (q, \epsilon)$, i.e., self-loops. Therefore, the RA also remains free of cycles except for self-loops. □

As stated earlier, the main purpose of the normal form is to ensure that any state can be reached backwards from a sink state. This property is captured in the following lemma.

Lemma 7. *Let $\mathcal{R}$ be an RA be in normal form, then for any state q, there exists a path $q, \ldots, q_{sink}$ such that q_{sink} is a sink state.*

Proof. Since the state space of $\mathcal{R}$ is finite and there exist no cycles except for self-loops, every state must have a path to a sink state. $\square$

The proof strategy is now as follows: In order to show completeness on the restricted class $\mathcal{C}$, we first provide a fixpoint characterization of the semantic state-based recovery equivalence. We show the correctness of the characterization by proving that the fixpoint is indeed the semantic state-based recovery equivalence. Using this characterization we can then proceed to show the main claim by showing that each iteration is captured by the syntactical state-based recovery equivalence.

Definition 14 *(Fixpoint characterization).* Define the fixpoint iteration $F : \mathbb{N} \to (Q \times Q)$ with

$$\begin{aligned} F_0 &:= \{(q,q) \mid q \in Q\} \cup \{(p,q) \mid p,q \in Q,\, p,q \text{ sink states with } p \sim_R q\} \\ F_n &:= F_{n-1} \cup \{(q_1,q_2) \mid q_1 \sim_R q_2, \forall B \notin DI(q_1) \cup DI(q_2) : \\ &\qquad \exists q_1', q_2' : \delta(q_1,B) = (q_1', rs), \delta(q_2,B) = (q_2', rs), \\ &\qquad (q_1', q_2') \in F_{n-1} \text{ or } q_1' = q_1 \text{ and } q_2' = q_2\} \end{aligned}$$

Let $Fix(F) := F_n$ denote the fixpoint of F such that $F_n = F_{n+1}$.

Intuitively, the iteration function F initially contains all states that are directly equivalent, either because they are equal or because they have no successors. That is, they are sink states. In each following iteration, pairs of states are then added in such a manner that all reachable successors of these pairs are already equivalent. The notion of reachability here also considers orthogonality, only considering transitions that are not disabled due to their guards being in the set of disabled inputs. If no further pairs of states can be added, then the fixpoint $Fix(F)$ is reached. Note that $Fix(F)$ is guaranteed to exist, since F is monotonic by definition, and the state space is finite. We prove with the following lemma that for the class of recovery automata consistent with respect to $\sim_R$, the fixpoint characterization indeed captures the recovery equivalence $\sim_R$.

Lemma 8. *If $\mathcal{R}$ is an RA in normal form and $\mathcal{R} \in \mathcal{C}$, then $Fix(F) = \sim_R$.*

Proof. It holds that $F_n \subseteq \sim_R$ for any n by definition of F_n. We now show that $\sim_R \subseteq F_n$ for some n. Let q_n denote a state such that any path of length at least n ends in a sink state. The existence of the sink state is guaranteed by Lemma 7 since $\mathcal{R}$ is in normal form by assumption. Proof by induction over n.

- Consider $n = 0$. Then q_0 and p_0 are sink states. If $q_0 \sim_R p_0$, then in total $(q_0, p_0) \in F_0$ by definition.
- Now assume if $p_i \sim_R p_j$ then $(p_i, q_j) \in F_k$ for some k for any $i, j < n$ (I). Consider some $p_n \sim_R q_m$ with $m \leq n$. Let $B \notin DI(p_n) \cup DI(q_m)$. Consider the following cases:
  - $\delta(q_m, B) = (q_m, rs)$ and $\delta(p_n, B) = (p_n, rs)$. Then by Definition 14, $(q_m, p_n) \in F_j$ is not required for any j.
  - $\delta(q_m, B) = (q_{m-1}, rs)$ and $\delta(p_n, B) = (p_{n-1}, rs)$. Since $\mathcal{R} \in \mathcal{C}$, it holds by Definition 12 that $p_{n-1} \sim_R q_{m-1}$. Hence by induction hypothesis (I) $(p_{n-1}, q_{m-1}) \in F_j$ for some j.

  In the following, we consider the cases where there is a single state change. Proof by induction over m.
  - Consider $m = 0$. Then $\delta(q_m, B) = (q_m, rs)$ and $\delta(p_n, B) = (p_{n-1}, rs)$. Since $\mathcal{R} \in \mathcal{C}$ it holds by Definition 12 that $p_{n-1} \sim_R q_m$. Hence by induction hypothesis (I) $(p_{n-1}, q_m) \in F_j$ for some j.
  - Consider $m > 0$. Assume if $p_n \sim_R p_i$ then $(p_n, q_i) \in F_j$ for some j for any $i < m$ (II). Assume without loss of generality that the state change occurs in q_m, otherwise swap p and q. Then $\delta(q_m, B) = (q_{m-1}, rs)$ and $\delta(p_n, B) = (p_n, rs)$. Since $\mathcal{R} \in \mathcal{C}$, it holds by Definition 12 that $p_n \sim_R q_{m-1}$. Hence, by induction hypothesis (II) $(p_n, q_{m-1}) \in F_j$ for some j.

  In all cases, this yields in total $(p_n, q_m) \in F_{j+1}$ for some j by Definition 14.

By induction principle this yields in total that if $p \sim_R q$ then $(p, q) \in F_n$ for some n. Since further by definition $F_n \subseteq F_{n+1}$ for any n, we hence obtain in total that there exists some n such that for any $p \sim_R q$ we have $(p, q) \in F_n$. $\square$

Using the fixpoint characterization, it is now easier to show what kind of states are recovery-equivalent by considering the individual fixpoint iterations and the pairs of equivalent states added in them respectively. With this it is now possible to move toward the main equivalence theorem. To this aim, we first show that generally any recovery-equivalent pair of states discovered by the fixpoint iteration, is also discovered by the syntactical recovery equivalence.

Lemma 9. *For any n it holds that* $F_n \subseteq \approx_R$.

Proof. Proof by induction over n.

- For $n = 0$ we have $F_0 \subseteq \approx_R$.
- Assume the hypothesis holds for some n. Let $(q_1, q_2) \in F_{n+1}$. If $(q_1, q_2) \in F_n$ then by induction hypothesis also $q_1 \approx_R q_2$. Now consider the case $(q_1, q_2) \notin F_n$. Since $(q_1, q_2) \in F_{n+1}$ we have $q_1 \sim_R q_2$ (I). Assume $q_1 \not\approx_R q_2$. Then there exists B with $\delta(q_1, B) = (q_1', rs_1)$, $\delta(q_2, B) = (q_2', rs_2)$, q_1, q_2 not orthogonal with respect to B and $rs_1 \neq rs_2$ or $q_1' \not\approx_R q_2'$.
  - Assume $rs_1 \neq rs_2$. Then $\delta^*(q_1, B) \neq \delta^*(q_2, B)$. Since q_1, q_2 are not orthogonal with respect to B, it holds that $B \notin DI(q_1) \cup DI(q_2)$. In total, this gives $q_1 \nsim_R q_2$. Contradiction to (I).
  - Assume $rs_1 = rs_2$. Then $q_1' \not\approx_R q_2'$. By definition of F_n, we also have $(q_1', q_2') \in F_n$. Hence by induction hypothesis, it also holds that $q_1' \approx_R q_2'$. Contradiction.

  In total, we thus obtain $q_1 \approx_R q_2$. □

By combining the individual preliminary results, the desired main result now follows.

Theorem 3. *If* $\mathcal{R}$ *is an RA in normal form and* $\mathcal{R} \in \mathcal{C}$*, then* $\sim_R \subseteq \approx_R$.

Proof. Since $\mathcal{R}$ is in normal form and $\mathcal{R} \in \mathcal{C}$ it holds that:

$$\begin{aligned} \sim_R &= Fix(F) && \text{(Lemma 8)} \\ &\subseteq \approx_R && \text{(Lemma 9)} \quad \square \end{aligned}$$

6. Implementation

A prototype of the described synthesis algorithm with the optimization rules has been implemented within the Virtual Satellite 4.0 framework [16]. Virtual Satellite 4.0 (VirSat) is an Eclipse-based framework intended for performing Model-Based Systems Engineering (MBSE) over the whole life cycle of space systems. VirSat provides a Generic Systems Engineering Language (GSEL) in which model extensions called *Conceptual Data Models* or short *concepts*, can be described. Each concept addresses a specific engineering aspect such as system decomposition, interface management, budgeting and so on. The developed prototype is a VirSat application called VirSat FDIR. It provides such a concept for modeling the FDIR engineering aspect. The tool and the concept were recently presented in [20]. The FDIR concept itself can be divided into the following sub-concepts:

- The Fault Concept, which focuses on modeling NdDFTs and DFTs.
- The Recovery Concept, which focuses on modeling recovery automata.
- The Requirements and Analysis Concept, which focuses on modeling analysis information and requirements referencing the result of an analysis.

The conceptual data model closely follows the objects that have been formally defined. For recovery automata, ϵ-loop transitions are not explicitly modeled. If some input B is undefined for some state, then it is assumed to be an ϵ-loop. The only addition not discussed in this paper is the requirements and analysis concept, which configures the analysis engines and contains their results as well as the requirements imposed on them.

The key feature of VirSat FDIR considered here is the synthesis algorithm. Moreover, it supports computing the actual metrics such as reliability after time t, mean time to failure, and fault tolerance. In addition to these analysis-focused features, the implementation also enables graphical modeling of NdDFTs and recovery automata. The goal of the software is to provide support for FDIR engineering throughout all the phases of spacecraft design, starting from the design of redundancy concepts and later on also supporting the design of on-board software FDIR. As input language, VirSat FDIR employs the Galileo file format [9], which can also be used to describe NdDFTs as they are syntactically the same as DFTs. The Galileo language has been implemented using XText [4]. All reliability data sets, state space sizes, benchmark times, and synthesized recovery automata in Section 7 have been generated using VirSat FDIR. In the following, we give some further details on the techniques applied in VirSat FDIR and their implementation.

6.1. Preprocessing

In order to simplify the handling of the diverse landscape of fault tree gates, fault trees are simplified by replacing gates that are expressible through other gates. For instance, AND and OR gates are replaced by corresponding k-VOTE gates, and priority gates are expressed by POR gates. With this approach, the set of gates that need to be considered is reduced to k-VOTE, SPARE, FDEP, RDEP, and POR.

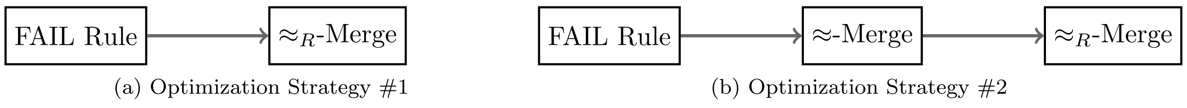


Fig. 16. Optimization workflows.

Table 1
MCS optimization results.

Equivalence Relation	#States	#Transitions	States Removed	Transitions Removed
–	991	7635	–	–
Trace Equivalence	67	559	93.24%	92.68%
Recovery Equivalence	41	378	95.86%	95.05%

6.2. *Canonical states*

Directly applying the presented Markov automaton semantics to NdDFTs generally yields huge state spaces. To keep the state space in a manageable size, we have applied a technique we call *canonical states*. When a basic event is added to a state, we also let all basic events fail that do not alter the future failure behavior of the fault tree. For example, consider a tree with an OR node whose sub-trees do not have outgoing edges to nodes outside of these sub-trees, except for the OR node. If now a BE fails and causes the OR node to fail, all BEs contained in the sub-tree of the OR node are set to failing as well. Since the OR node has already failed, failures of the contained BEs does not affect the failure behavior of the fault tree. A Markov automaton state in which the maximum number of such BEs have failed is called a *canonical* state. We transform all states upon their generation into their respective canonical form.

6.3. *Optimization workflow*

Performing the orthogonal merge requires the computation of the disabled inputs on all states. As the initial recovery automaton, which is extracted from a Markov automaton, can be very big, computing the set of disabled inputs for every state requires the computation of many set intersections. In addition to directly optimizing a recovery automaton with the Recovery Equivalence relation, we therefore also consider an optimization strategy where the recovery automaton is first optimized using classical trace equivalence $\approx$ (Definition 5). The resulting recovery automaton with reduced state space is further optimized using the $\approx_R$ relation (Definition 7). In addition, the FAIL rule is applied in the beginning in both workflows. The resulting optimization workflows are visualized in Fig. 16.

7. Case studies

In order to assess the presented techniques, we applied the synthesis methodology including the newly described optimization rules to case study examples. We also compared the performance of the two presented optimization strategies. All experiments were conducted with a Intel i7-6600U CPU and 16 GB of RAM. Moreover, all timed benchmarks were taken by measuring the average over 100 runs.

7.1. *Multiprocessor computing system*

Target system We consider the literature example of a Multiprocessor Computing System (MCS) based on the model given in [6]. The MCS consists of two main components: the Bus and the Computing Module (CM). The CM is hot-redundant and consists of two further CMs: CM_1 and CM_2. Each of these CMs requires a disk, a processor, and a memory unit. Each CM has a hot-redundant backup disk. Furthermore, a shared redundant memory unit MS is available to the entire CM in case that their own memory unit fails. Finally, both processors are powered by a common power source PS. This power source is again hot-redundant and consists of the two power units PS_1 and PS_2. Fig. 17 shows a NdDFT that describes the MCS.

Experimental results The described synthesis algorithm was performed to obtain a recovery automaton from the described NdDFT. The RA was then optimized by merging trace-equivalent and recovery-equivalent states and by eliminating redundant transitions.

Table 1 shows the results after minimizing the synthesized RA. Observe that initially the RA contains a large number of states and transitions. After performing the partition refinement algorithm based on trace equivalence, the number of states and transitions is significantly reduced. After performing the partition refinement and merging non-trace-equivalent states according to the described optimization rules, it can be observed that the number of states and transitions is altogether reduced by 95.86% and 95.05%, respectively. Thus, merging non-trace-equivalent states additionally reduces the number of states obtained by merging trace-equivalent states by 38.81%, and the number of transitions by 32.38%. This indicates the

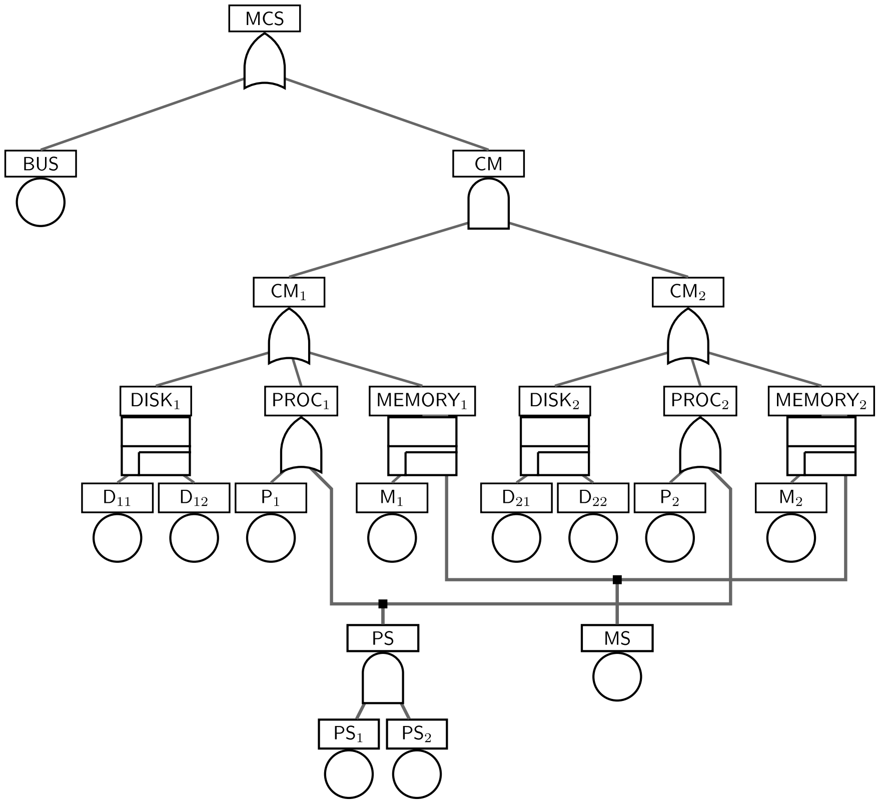


Fig. 17. NdDFT of the Multiprocessor Computing System.

Table 2
MCS benchmark results.

Equivalence Relation	Time (ms)
Trace Equivalence	204
Recovery Equivalence (Strategy #1)	494
Recovery Equivalence (Strategy #2)	339

effectiveness of the proposed approach to consider cases where non-trace-equivalent states can be merged to obtain an equivalent recovery automaton having the same behavior.

Table 2 shows the benchmarked times to perform the optimization. First of all, it can be seen that the overhead caused by the optimization procedure is in general negligible. However, comparing the optimization strategies amongst each other, big gaps in performance can be identified. With an increase of computation time by 142%, employing the Recovery Equivalence (Strategy #1) relation clearly imposes a burden on the optimization process. However, by applying the Recovery Equivalence relation with optimization strategy #2, a major part of the burden can be lifted, reducing the overhead to 66%.

7.2. Nested spare system

Target system All examples so far have only considered spare gates that have basic events as children. However, as initially introduced, there is no such syntactical restriction on NdDFTs. Going further, fault trees are even allowed to have complex sub-trees as spares which in return also contain spare gates. In the following, a system with nested spares is considered by modifying the previous MCS literature use case. The NdDFT is obtained by using the CM node as the new root and by replacing instances of AND gates by instances of SPARE gates. We also consider a third computing module CM_3 to increase the state space. Fig. 18 shows a NdDFT that describes the modified MCS.

Experimental results Just as in the previous experiment, an optimized recovery automaton has been obtained from the modified MCS NdDFT by applying the synthesis algorithm and then the optimization techniques. The new results are shown in Table 3.

First of all, it can be observed that the non-optimized state space is significantly smaller than that of the original MCS, despite adding a third Computing Module. This is due to the modification of turning the AND gates into SPARE gates, which

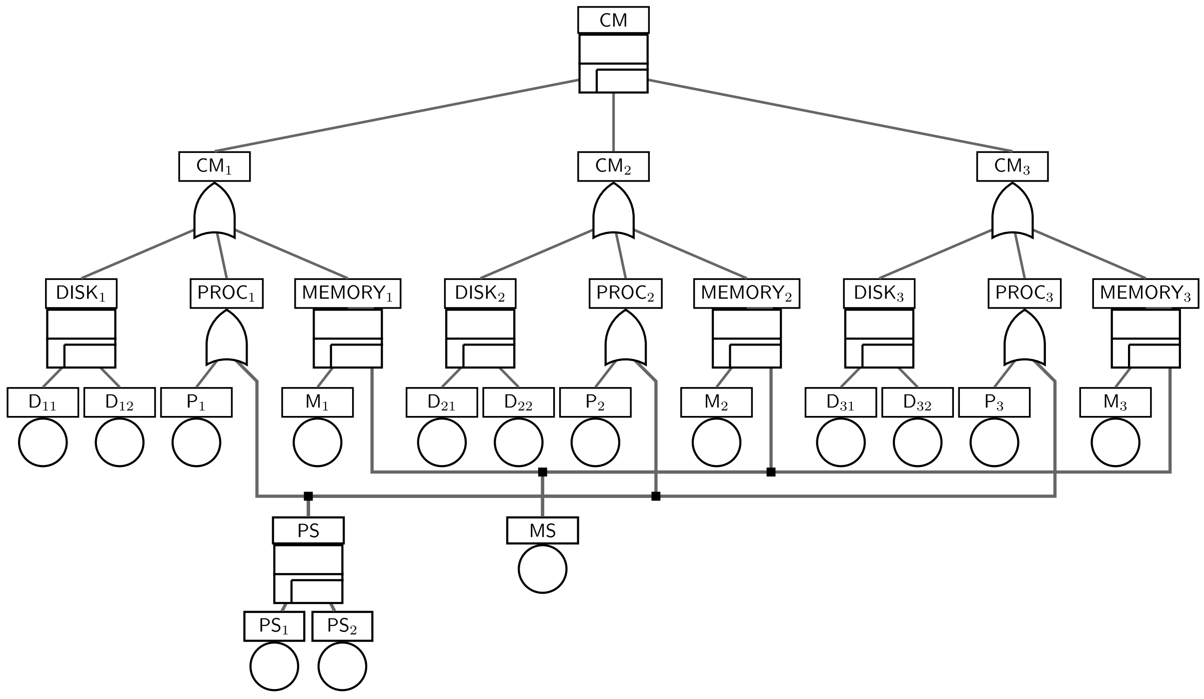


Fig. 18. NdDFT of the modified Multiprocessor Computing System.

Table 3
Optimization results of modified MCS.

Equivalence Relation	#States	#Transitions	States Removed	Transitions Removed
–	229	1216	–	–
Trace Equivalence	173	916	24.45%	24.67%
Recovery Equivalence	158	862	31.00%	29.11%

Table 4
Benchmark results of modified MCS.

Equivalence Relation	Time (ms)
Trace Equivalence	236
Recovery Equivalence (Strategy #1)	416
Recovery Equivalence (Strategy #2)	349

greatly reduces the number of possible interleavings of fault events. For instance, in the beginning, only child BEs of the primary of CM_1 can fail, as all other BEs are inactive. However, while the initial state space has been decreased drastically, it can also be observed that the optimization of the RA yielded far less radical minimization results. After performing the partition refinement algorithm based on the trace equivalence definition, the number of states and transitions is reduced by only 24.45% and 24.67% respectively. After performing the partition refinement and merging non-trace-equivalent states according to the described optimization rules, it can be observed that the number of states and transitions is altogether reduced by 31% and 29.11%, respectively. Thus, merging non-trace-equivalent states additionally reduces the number of states obtained by merging trace-equivalent states by 8.67% and the number of transitions by 5.9%. As can be seen, employing Recovery Equivalence for optimization yields a decent improvement in the state space reduction, but in this case falls behind the aggressive improvement that could be observed in the prior case study.

Table 4 shows the benchmarked times to perform the optimization. The results show that contrary to the state space decrease, the optimization time has roughly stayed the same. This is likely due to the less effective state space reduction, leading to more iterations for the partition refinement algorithm. Again, the time to perform the actual optimization is in general short but varies strongly among the different optimization strategies. Employing the Recovery Equivalence relation yields an increase in computation time of 76%. The improved workflow, on the other hand, gives an increase of about 47%.

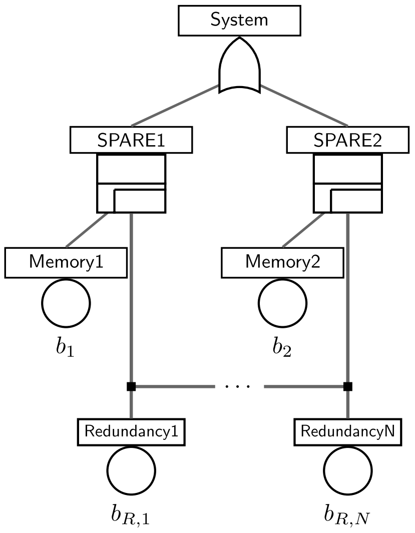


Fig. 19. Memory system with N redundancies.

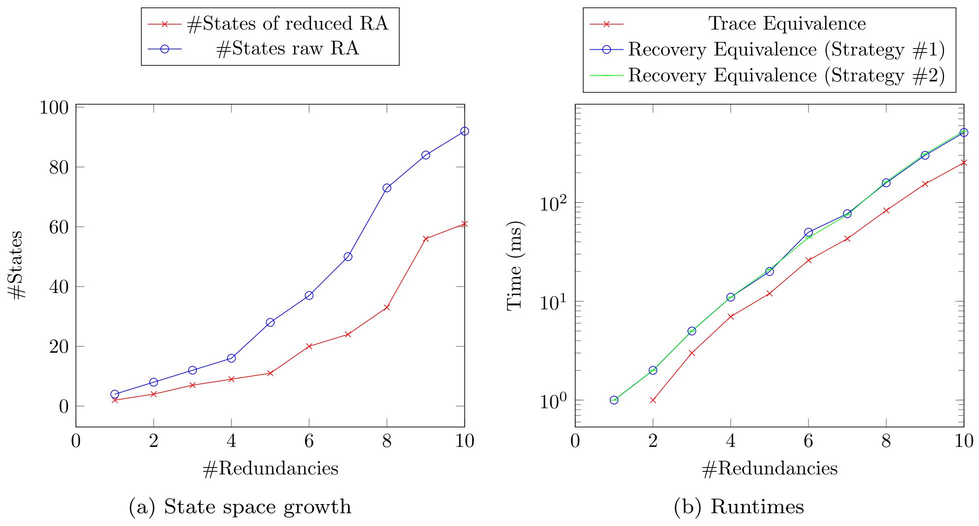


Fig. 20. Results for RA of memory system with N redundancies.

7.3. Memory system with N redundancies

Target system To assess the scalability of state space reduction for recovery automata in terms of increasing DFT complexity, we consider a family of DFTs based on the previous memory system use case given in Fig. 4. The model family is depicted in Fig. 19. As before, the system consists of two main memory units Memory1 and Memory2. However, instead of a fixed number of redundant memory systems, they now share a variable pool of cold redundancies of size N. Also, the FDEPs modeling the power system have been removed.

Experimental results Fig. 20a shows how the state space sizes increase with varying number of redundancies N for both the raw RA as extracted from the MA and the finally resulting minimized RA. It can be seen that the RA state space grows significantly slower but still at an exponential pace. However, it can also be seen that the state space reduction remains consistent over the course of the increasing number of redundancies. The benchmarked timing results are captured in Fig. 20b. Note that the y-axis is scaled logarithmically. In this case, both optimization strategies for the Recovery Equivalence relation yield nearly identical performance results.

8. Conclusions and future work

In this paper, we investigated the problem of optimizing recovery automata that represent recovery strategies synthesized from NdDFTs. New algorithms to minimize an RA by additionally eliminating some non-trace-equivalent states and redundant transitions were provided. In particular, we extended the notion of recovery equivalence between states by introducing the concept of orthogonal states and a rule for merging them. In addition, we introduced the concept of FAIL states and a rule for merging them with predecessor states. A formal proof showing that an equivalent RA is produced for each case was given. This proof was complemented by an analysis of the completeness of our approach, showing that this property is generally not ensured. A class of RA for which the approach is complete was also characterized. Case studies using the described approach were provided, and the evaluated results showed that it allows to obtain a more efficient implementation of recovery strategies for FDIR than solely eliminating trace-equivalent states.

The optimization of recovery automata as presented in Section 5 is based on the observation that permanent errors in fault trees cannot occur twice. In the future, we would like to extend the model to deal with fault trees that exhibit both transient and repairable faults and to investigate how the merging rules can be transferred and extended. In fact, first steps in this direction have already been undertaken. In [19], the specific choice of sets of error events as input symbols is generalized by a symbolic input alphabet enriched by a disabling relation. The latter is a binary relation on input symbols that specifies that the occurrence of an input excludes the later occurrence of certain other inputs. In our setting, it relates two sets of error events whenever they are not disjoint.

Moreover, we would like to investigate a generalization of the optimization formalisms, aiming to achieve the desired completeness property.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Analysis Techniques for System Reliability-Procedure for Failure Mode and Effects Analysis (FMEA), Technical Committee and others, Geneva, Switzerland, 2006.
[2] Fault Tree Analysis (FTA), International Electrotechnical Commission, Geneva, Switzerland, 2006.
[3] M. Beccuti, G. Franceschinis, D. Codetta-Raiteri, S. Haddad, Computing optimal repair strategies by means of NdRFT modeling and analysis, Comput. J. 57 (2014) 1870–1892, https://doi.org/10.1093/comjnl/bxt134.
[4] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing Ltd, 2016.
[5] B. Bittner, M. Bozzano, A. Cimatti, R. De Ferluc, M. Gario, A. Guiotto, Y. Yushtein, An integrated process for FDIR design in aerospace, in: Model-Based Safety and Assessment, in: LNCS, vol. 8822, Springer, 2014, pp. 82–95.
[6] A. Bobbio, L. Portinale, M. Minichino, E. Ciancamerla, Improving the analysis of dependable systems by mapping fault trees into Bayesian networks, Reliab. Eng. Syst. Saf. 71 (2001) 249–260, https://doi.org/10.1016/S0951-8320(00)00077-6.
[7] D. Codetta-Raiteri, L. Portinale, Dynamic Bayesian networks for fault detection, identification, and recovery in autonomous spacecraft, IEEE Trans. Syst. Man Cybern. Syst. 45 (2015) 13–24, https://doi.org/10.1109/TSMC.2014.2323212.
[8] J.B. Dugan, S.J. Bavuso, M.A. Boyd, Dynamic fault-tree models for fault-tolerant computer systems, IEEE Trans. Reliab. 41 (1992) 363–377, https://doi.org/10.1109/24.159800.
[9] J.B. Dugan, K.J. Sullivan, D. Coppit, Developing a low-cost high-quality software tool for dynamic fault-tree analysis, IEEE Trans. Reliab. 49 (2000) 49–59.
[10] E. Edifor, M. Walker, N. Gordon, Quantification of priority-or gates in temporal fault trees, in: International Conference on Computer Safety, Reliability, and Security, in: LNCS, vol. 7612, Springer, 2012, pp. 99–110.
[11] C. Eisentraut, H. Hermanns, L. Zhang, On probabilistic automata in continuous time, in: IEEE Symposium on Logic in Computer Science, IEEE, 2010, pp. 342–351.
[12] D. Guck, H. Hatefi, H. Hermanns, J.P. Katoen, M. Timmer, Modelling, reduction and analysis of Markov automata, in: Quantitative Evaluation of Systems, in: LNCS, vol. 8054, Springer, 2013, pp. 55–71.
[13] J. Hopcroft, An n log n algorithm for minimizing states in a finite automaton, in: Theory of Machines and Computations, Elsevier, 1971, pp. 189–196.
[14] S. Junges, D. Guck, J.P. Katoen, M. Stoelinga, Uncovering dynamic fault trees, in: 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN, IEEE, 2016, pp. 299–310.
[15] G.A. Kildall, A unified approach to global program optimization, in: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, 1973, pp. 194–206.
[16] C. Lange, J.T. Grundmann, M. Kretzenbacher, P.M. Fischer, Systematic reuse and platforming: application examples for enhancing reuse with model-based systems engineering methods in space systems development, Concurr. Eng. 26 (2018) 77–92, https://doi.org/10.1177/1063293X17736358.
[17] G. Mealy, A method for synthesizing sequential circuits, Bell Syst. Tech. J. 34 (1955) 1045–1079, https://doi.org/10.1002/j.1538-7305.1955.tb03788.x.
[18] L. Mikaelyan, S. Müller, A. Gerndt, T. Noll, Synthesizing and optimizing FDIR recovery strategies from fault trees, in: FTSCS 2018, in: CCIS, vol. 1008, Springer, 2019, pp. 37–54.
[19] M. Mileva, Minimising Mealy Machines with Dependent Inputs, Bachelor thesis, RWTH Aachen University, 2019, http://www-i2.informatik.rwth-aachen.de/pub/index.php?type=download&pub_id=1751.
[20] S. Müller, A. Gerndt, Towards a conceptual data model for fault detection, isolation and recovery in Virtual Satellite, in: SECESA 2018, European Space Agency, 2018, https://elib.dlr.de/122061/.
[21] S. Müller, A. Gerndt, T. Noll, Synthesizing FDIR recovery strategies from non-deterministic dynamic fault trees, in: 2017 AIAA SPACE Forum, vol. AIAA 2017-5163, American Institute of Aeronautics and Astronautics, 2017.
[22] S. Müller, A. Gerndt, T. Noll, Synthesizing failure detection, isolation, and recovery strategies from nondeterministic dynamic fault trees, J. Aerosp. Inform. Syst. (2018) 1–9, https://doi.org/10.2514/1.I010669.

[23] D.C. Raiteri, L. Portinale, ARPHA: an FDIR architecture for Autonomous Spacecrafts based on Dynamic Probabilistic Graphical Models, Technical Report TR-INF-2010-12-04-UNIPMN, Computer Science Institute, Università del Piemonte Orientale, Vercelli, Italy, 2010, http://www.di.unipmn.it/TechnicalReports/TR-INF-2010-12-04-UNIPMN.pdf.

[24] E. Ruijters, D. Guck, P. Drolenga, M. Stoelinga, Fault maintenance trees: reliability centered maintenance via statistical model checking, in: 2016 Annual Reliability and Maintainability Symposium, RAMS, IEEE, 2016, pp. 1–6.

[25] E. Ruijters, M. Stoelinga, Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools, Comput. Sci. Rev. 15–16 (2015) 29–62, https://doi.org/10.1016/j.cosrev.2015.03.001.

[26] W.E. Vesely, F.F. Goldberg, N.H. Roberts, D.F. Haasl, Fault Tree Handbook, Technical Report, Nuclear Regulatory Commission, Washington, DC, 1981, https://www.osti.gov/biblio/5762464-fault-tree-handbook.

[27] M. Volk, S. Junges, J.P. Katoen, Advancing dynamic fault tree analysis-get succinct state spaces fast and synthesise failure rates, in: International Conference on Computer Safety, Reliability, and Security, in: LNCS, vol. 9922, Springer, 2016, pp. 253–265.

[28] A. Wander, R. Förstner, Innovative fault detection, isolation and recovery strategies on-board spacecraft: state of the art and research challenges, in: Deutscher Luft- und Raumfahrtkongress 2012, German Soc. for Aeronautics and Astronautics – Lilienthal-Oberth e.V., Bonn, Germany, 2013, https://www.dglr.de/publikationen/2013/281268.pdf.