Formal Verification of Rewriting Rules for Dynamic Fault Trees

Yassmeen Elderhalli[1(✉)], Matthias Volk[2], Osman Hasan[1], Joost-Pieter Katoen[2], and Sofiène Tahar[1]

[1] Electrical and Computer Engineering, Concordia University, Montréal, Canada
{y_elderh,o_hasan,tahar}@ece.concordia.ca
[2] Software Modeling and Verification, RWTH Aachen University, Aachen, Germany
{matthias.volk,katoen}@cs.rwth-aachen.de

Abstract. Dynamic Fault Trees (DFTs) model the failure behavior of systems dynamics. Several rewriting rules have been recently developed, which allow the simplification of DFTs prior to a formal analysis with tools such as the Storm model checker. To ascertain the soundness of the analysis, we propose to formally verify these rewriting rules using higher-order-logic (HOL) theorem proving. We first present the formalization in HOL of commonly used DFT gates, i.e., AND, OR and PAND, with an arbitrary number of inputs. Then we describe our formal specification of the rewriting rules and the verification of their intended behavior using the HOL4 theorem prover.

Keywords: Dynamic Fault Trees · Rewriting rules · Theorem proving · HOL4

1 Introduction

Dynamic Fault Trees (DFTs) graphically model the dynamically changing failure dependencies between system components [15,16]. The modeling starts by a top event that represents an undesired event, like the failure of a system or subsystem. Then, the different relationships between the system basic events that lead to the failure of the top event are modeled using DFT gates. DFTs are more suitable to model real-world situations that cannot be captured using static fault trees (SFTs). For example, DFTs models have been used to provide the safety analysis for autonomous cars [8].

DFTs are directed acyclic graphs (DAG) with typed nodes (AND, OR, etc.). Successors of a node v in the DAG are *inputs* of v. Some commonly used DFT elements are shown in Fig. 1. Nodes without inputs are *basic events* (BE, Fig. 1(a)) that represent atomic components, which can fail according to a failure distribution. Special cases of BEs are *constant failed* elements (CONST($\top$), Fig. 1(b)), which are always failed and *constant fail-safe* elements (CONST($\bot$), Fig. 1(c)),

This work is partially supported by the DFG RTG 2236 UnRAVeL.

P. C. Ölveczky and G. Salaün (Eds.): SEFM 2019, LNCS 11724, pp. 513–531, 2019.
https://doi.org/10.1007/978-3-030-30446-1_27

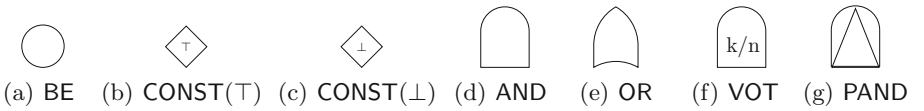


Fig. 1. Some DFT elements

which can never fail. DFT *gates* are nodes with inputs and are used to model the state dependencies and redundancies among system components. Some commonly used DFT gates include SFT gates (AND, OR and VOT-gates) as well as the Priority-AND (PAND) DFT gate. The output event of the AND-gate (Fig. 1(d)) fails when both input events fail. The OR-gate (Fig. 1(e)) requires that at least one of its input events fails for the output event to fail. The output of the $\mathsf{VOT}_{\mathrm{k}}$-gate ($k$ out of n gate) (Fig. 1(f)) fails when at least k out of the n inputs fail. The PAND-gate (Fig. 1(g)) acts in a similar way to the AND-gate, i.e., it requires that both input events fail. However, an additional condition is needed, where the inputs should fail in sequence, usually from left to right. There are also other DFT gates that are used to model the dynamic behavior in systems, like the Functional-DEPendency (FDEP) and spare gates. In this paper, we only consider DFTs with AND, OR, VOT and PAND-gates.

Traditionally, DFTs are analyzed quantitatively by converting the given DFT model into a Markov chain (MC) [1,3,17], where the latter can be analyzed analytically or using simulation. Recently, an algebra has been proposed to provide the analysis of DFTs analytically without the need to use MC models [12]. In the algebraic approach, temporal operators are defined to capture the failure dependency between system components. The DFT gates are modeled using these temporal operators and their probabilities of failure are expressed based on these operators. Moreover, the DFT algebra provides several simplification properties that allow reducing the structure of a given DFT for a simpler analysis.

In order to ensure a complete and sound analysis, formal methods have also been explored for analyzing DFTs. Probabilistic model checkers, such as Storm [2], have been used for the probabilistic analysis of DFTs via MCs. For example, Storm supports the analysis of DFTs, among other probabilistic models, and allows the verification of the probability of failure and the Mean-Time-To-Failure (MTTF) of the top event of a given DFT. The scalability of this analysis can be significantly improved by DFT rewriting rules [10] that facilitate simplifying a DFT before analysis. Simplification of the DFT is achieved by transforming the underlying graph of the DFT according to the rewrite rules. Experimental evaluation in [10] showed that rewriting heavily improves the performance of the DFT analysis. For example, while originally 68% of the 183 DFTs in [10] could be solved within 2 h, applying the rewriting beforehand allowed to solve 95% of the DFTs. Moreover, the total analysis time was reduced from 41 h to 18 h when using rewriting. Simplifying DFTs by rewriting enables the

analysis of DFTs that could not be analyzed before, and can lead to speed-ups and memory savings of up to two orders of magnitude [10].

The rewrite rules are generic for n-ary gates and can be implemented in any tool that supports DFT analysis. Proving the correctness of the rewrite rules as done in [11] is an involved manual and error-prone process. To the best of our knowledge, a rigorous, mechanically checkable proof of correctness of these rewriting rules has not been done. Thus, their usage in a formal analysis raises soundness concerns especially when dealing with the analysis of safety-critical systems, like transportation or healthcare. On the other hand, higher-order logic (HOL) theorem proving has been recently used to formalize DFT gates and operators [6] based on the algebra presented in [12]. Several simplification theorems are formally verified using the HOL4 theorem prover [9], which enable formally verifying a reduced form of a DFT. Moreover, the probabilistic behaviors of DFT gates are formally verified based on the HOL4 probability and Lebesgue integral theories [13,14]. However, this formalization does not support n-ary gates, which are required to model generic failure scenarios. In addition, the VOT-gate has not been formalized in HOL.

In this paper, we propose to use the recent HOL DFT formalization to verify the DFT rewriting rules of [10] using the HOL4 theorem prover. This requires extending the DFT gates definitions in [6] for an arbitrary number of inputs and defining the VOT-gate. Our main contributions are summarized as follows:

- Higher-order logic formalization of AND, OR, PAND and $\mathsf{VOT}_{\mathrm{k}}$ (k out of n) gates for arbitrary number of inputs. This allows a formal reasoning about generic DFT constructs.
- A mechanized verification in HOL4 of the correctness of the DFT rewrite rules of [10] that are concerned with DFTs with AND, OR, VOT and PAND-gates. This proves that all these rules preserve reliability and MTTF.

These contributions provide the assurance of correctness of the rewrite rules and thus adds the confidence to tools, that exploit these rules in their DFT analysis.

The rest of the paper is structured as follows: Sect. 2 describes the DFT rewrite rules. We review the HOL4 DFT theory (library) in Sect. 3. In Sect. 4, we present the HOL formalization of n-ary gates. The formal verification details of the rewrite rules are presented in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 DFT Rewrite Rules

In the following, we recap the rewrite rules for DFTs as presented by Junges *et al.* [10]. The simplification of DFTs is performed by graph rewriting [4] on the underlying graph of the DFT. We represent a DFT as a labelled graph by extending the induced graph with labels encoding the type of the DFT element and the ordering of the inputs. The graph transformation on the labelled graph is performed by applying a chain of rewrite rules.

2.1 Rewrite Framework

A rewrite rule is specified by two (sub-)DFTs: the left-hand side capturing the (sub-)DFT before applying the rewrite rule and the right-hand side depicting the resulting (sub-)DFT after the graph rewrite. An example of a rewrite rule is given in Fig. 2. The rule depicts the subsumption of OR-gates by AND-gates.

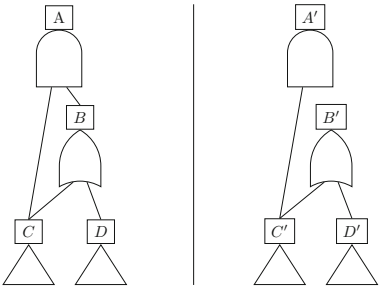
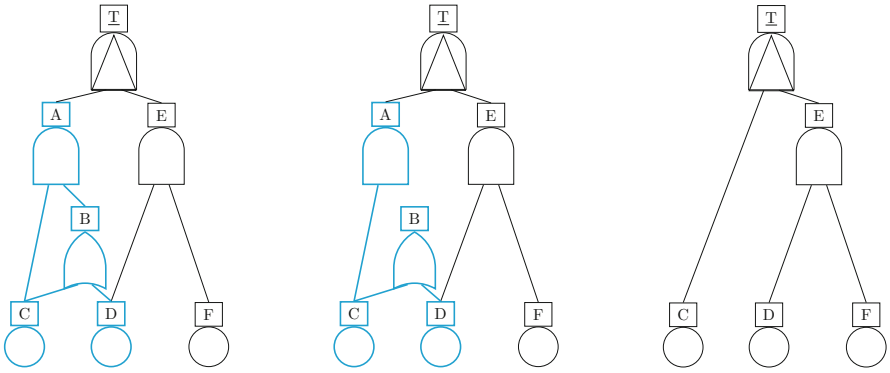


Fig. 2. Subsumption of OR-gates by AND-gates [10, Rewrite rule 8]



(a) Original DFT (b) DFT after rewrite step (c) Final DFT

Fig. 3. Example application of rewrite rule (Color figure online)

A rewrite rule can be applied whenever a (sub-)DFT can be matched with the left-hand side of the rule. Elements represented by a triangle in the rewrite rule match every gate type. Matched elements might have additional ingoing and outgoing edges not matched by the rewrite rule. These edges are retained during the rewriting step. Applying a rewrite rule replaces the matched part with the right-hand side of the rule. All non-matched parts remain unchanged during the rewriting step. Note that in general, rewrite rules might lead to inconsistent

graphs with dangling edges or DFTs that are no longer well-formed (e.g., cyclic DFTs). In these cases, the rewrite rule cannot be applied. It is important to note also that most of the rewrite rules can also be applied from right to left.

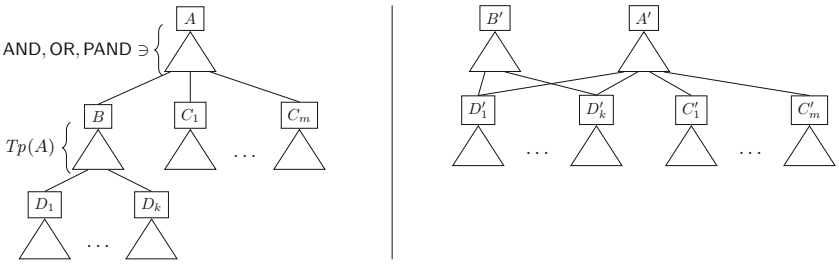


Fig. 4. Left-flattening of gates [10, Rewrite rule 5]

An example application of the given subsumption rule is depicted in Fig. 3. Figure 3(a) depicts the original DFT used as input. The subsumption rule from Fig. 2 can be applied and the matched sub-DFT is highlighted in blue. Applying the rule removes the connection between AND-gate A and OR-gate B and yields the rewritten DFT in Fig. 3(b). Further simplification by applying additional rewrite rules results in the final DFT in Fig. 3(c). Using the rewrite rules leads to a simpler DFT, which is considerably smaller—and easier to understand.

During rewriting multiple rules might be applicable for the current DFT or different sub-DFTs match the left-hand side of a rewrite rule. The sequence of rewrite steps is chosen by a rewrite strategy. As the rewrite framework is not confluent, the strategy heavily influences the size of the resulting DFTs and a heuristic approach is used. For further details, see [10].

2.2 Rewrite Rules

In the following we consider 22 rules of the 29 rewrite rules given in [10]. Of the remaining 7 rules, one rule gives the Shannon expansion for VOT_k-gates, which deals with variables as Boolean, whereas generally DFTs, as formalized in HOL, treat variables as real numbers representing time to failure functions. The other 6 rules apply to FDEPs and SPAREs; both gate types are not considered here. We recap a selection of the rewrite rules and use the same rule enumeration as in [10, Sect. 5.3].

General Rewrite Rules. The first rewrite rules 1–7 consider structural identities such as commutativity of static gates, removal of gates with a single successor or no predecessor, and left-flattening of gates. As an example, the rewrite rule for left-flattening is given in Fig. 4. The rule can only be applied if the top element of the (sub-)DFT is an AND-, OR- or PAND-gate, and the first input is of the same

gate type as the top element ($Tp(B) = Tp(A)$). Applying the left-flattening rule adds the inputs of B as first inputs of A. Gate B is not removed as it might still have connections to other parts of the DFT.

Rules 8–10 capture standard axioms from Boolean algebra on the static gates such as subsumption of OR-gates by AND-gates (cf. Fig. 2).

DFTs containing constant failed CONST($\top$) or constant fail-safe CONST($\bot$) events can lead to large simplifications as often complete sub-DFTs can be evaluated to constant. Rules 11–14 specifically consider constant elements and we exemplary present the rewrite rule for AND-/PAND-gates with CONST($\bot$) inputs in Fig. 5. If at least one of the inputs of an AND-/PAND-gate is fail-safe, it is impossible for the gate to fail and therefore it can be set to fail-safe as well.

Encoding of VOT-gates by OR-/AND-gates is given in rewrite rules 15–16.

Rewrite Rules for PAND-gates. So far, the rewrite rules mostly captured simplifications of static gates, which are based on the corresponding properties in Boolean algebra. The remaining rules 18–23 consider PAND-gates where the order of failures is crucial. As an example, consider the rewrite rule for conflicting PAND-gates with independent successors in Fig. 6. PAND-gate D_1 requires that input B fails strictly before C or simultaneously with C. If C fails strictly before B, D_1 becomes fail-safe. D_2 requires the opposite behavior. If both elements B and C are independent, they will not fail simultaneously. Thus, either PAND-gate D_1 or D_2 will become fail-safe. As the PAND-gates can never both fail, A is fail-safe and can be replaced by CONST($\bot$).

Note that the rewrite rule can only be applied if B and C are independent—and at most one input is CONST($\top$). Otherwise, a common cause failure can let both B and C fail simultaneously, both PAND-gates fail and A fails as well. The independence assumption in this rewrite rule is a *context restriction*, which prevents the application of the rule for certain DFTs.

2.3 Non-structural Rules

There are two additional rules that are not present in the rewrite framework as they go beyond structural rules and are not captured by graph transformations.

Removing BEs. The BEs that have no connection to other DFT elements (and are not the top level element) are called *dispensable*. Dispensable BEs can be removed from the DFT as they do not influence the analysis results. An example is given in Fig. 7. In the original DFT in Fig. 7(a), BE C is dispensable and can be removed yielding the DFT in Fig. 7(b).

Merging BEs. In our analysis we are only interested in the reliability or MTTF of the top level element. The state of other elements is not important for this analysis. Thus, we can simplify a DFT by merging multiple BEs into a single BE. Consider the example DFT in Fig. 7(b). Both BEs A and B have an exponential failure distribution with failure rates λ_A and λ_B, respectively. The failure

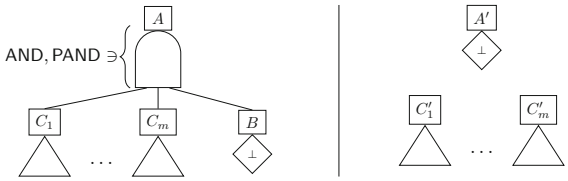


Fig. 5. AND-/PAND-gate with CONST(⊥) successor [10, Rewrite rule 13]

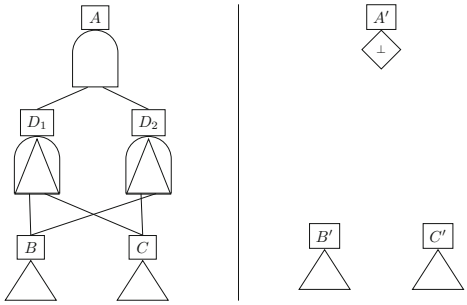
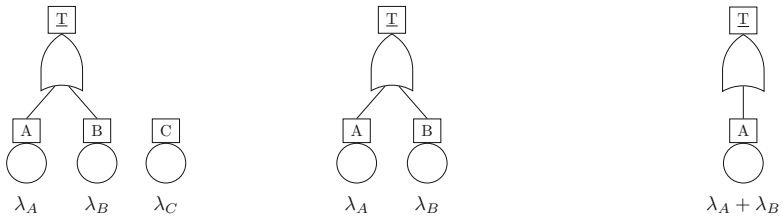


Fig. 6. Conflicting PAND–gates with independent successors [10, Rewrite rule 19]



(a) Original DFT (b) DFT after removal of BE C (c) DFT after merging of BEs

Fig. 7. Example application of non-structural rules

distribution of an OR-gate is the minimum over its inputs and is exponentially distributed as well. Thus, we can replace multiple BEs $A_1, \ldots, A_n$ under an OR-gate by a single BE A' with failure rate $\lambda_{A'} = \sum_{i=1}^{n} \lambda_{A_i}$. In our example, merging both BEs leads to the final DFT in Fig. 7(c). The resulting OR-gate with a single input can be simplified further by applying the rewrite framework.

After presenting the details of DFT rewrite rules, in the sequel, we present our efforts in formally verifying them using HOL theorem proving. For some of these rules, such as Rule 5, it is required to formally model DFT gates for arbitrary number of inputs. In the next sections, we first review the DFT theory developed in HOL4 and then introduce the new HOL definitions of n-ary gates.

3 DFT Theory in HOL4

DFTs have been formalized using the HOL4 theorem prover [6] based on the algebra presented in [12]. In this algebra, gates are modeled based on the time of failure of their outputs. Inputs of a DFT represent the time-to-failure functions of systems components. Therefore, in the DFT formalization, these functions are defined as lambda abstracted functions that allow them to be treated later as random variables for conducting the probabilistic analysis of DFTs. Identity elements and temporal operators are introduced to allow expressing and manipulating the structure function of the top level element of a given DFT. Their mathematical expressions and HOL formalization are presented in Table 1, where `PosInf` is the HOL4 representation of $+\infty$.

The Always identity element is used to model an event that fails from time 0, whereas the Never element models an event that fails at $+\infty$, i.e., it can never fail. These two elements are necessary in the simplification process of DFTs, when there are input events that are fail-safe ($\mathsf{CONST}(\bot)$) or have already failed ($\mathsf{CONST}(\top)$). Therefore, these functions that represent the inputs and outputs of DFT gates return extended-real numbers (HOL4 extreal theory), which are real numbers and $\pm\infty$. Three temporal operators are introduced in [12] to model the failure dependency among system components. The Before operator ($\lhd$) models a situation where one system component fails before the other. This operator accepts two inputs and its output fails when the first input fails before the second, otherwise it can never fail. The Simultaneous operator (Δ) requires that both inputs fail at the same time for its output to fail. If this condition does not hold, then the output of this operator fails at $+\infty$. Finally, the output of the Inclusive Before operator ($\trianglelefteq$) fails when the first input fails before or at the same time of the second input, otherwise it does not fail.

The AND ($\cdot$) and OR ($+$) gates are similar to the ones used in SFTs. However, it is required to define them in a way compatible with the rest of the definitions of DFT gates. Table 2 [6] lists the formal definitions of these gates, where `max` and `min` are HOL4 functions that return the maximum and minimum values of their input arguments, respectively. The output of the AND-gate (Fig. 1(d)) is modeled using the maximum (`max`) time of failure of the inputs. The OR-gate

Table 1. Definitions of identity elements and temporal operators

Element/Operator	Mathematical expression	Formalization
`Always element`	$d(ALWAYS) = 0$	`⊢ ALWAYS = (λs. (0:extreal))`
`Never element`	$d(NEVER) = +\infty$	`⊢ NEVER = (λs. PosInf)`
`Before`	$d(A \lhd B) = \begin{cases} d(A), & d(A) < d(B) \\ +\infty, & d(A) \geq d(B) \end{cases}$	`⊢ ∀ A B. D_BEFORE A B = (λs. if A s < B s then A s else PosInf)`
`Simultaneous`	$d(A \Delta B) = \begin{cases} d(A), & d(A) = d(B) \\ +\infty, & d(A) \neq d(B) \end{cases}$	`⊢ ∀ A B. D_SIMULT A B = (λs. if A s = B s then A s else PosInf)`
`Inclusive Before`	$d(A \trianglelefteq B) = \begin{cases} d(A), & d(A) \leq d(B) \\ +\infty, & d(A) > d(B) \end{cases}$	`⊢ ∀ A B. D_INCLUSIVE_BEFORE A B = (λs. if A s ≤ B s then A s else PosInf)`

Table 2. DFT gates

Gate	Mathematical expression	Formalization
AND	$d(A \cdot B) = max(d(A), d(B))$	⊢ ∀ A B. D_AND A B = (λs. max (A s)(B s))
OR	$d(A + B) = min(d(A), d(B))$	⊢ ∀ A B. D_OR A B = (λs. min (A s)(B s))
PAND	$d(Q_{PAND}) = \begin{cases} d(B), & d(A) \leq d(B) \\ +\infty, & d(A) > d(B) \end{cases}$	⊢ ∀ A B. PAND A B = (λs. if A s ≤ B s then B s else PosInf)

(Fig. 1(e)) requires that at least one of its input events fails. Therefore, the time of failure of its output is modeled using the minimum (`min`) time of failure of its inputs. The PAND-gate (Fig. 1(g)) is modeled using the `extreal` comparison operator ($\leq$) and if statements. The time of failure of its output equals the time of the second input if the first input fails before or at the same time of the second input, otherwise, the output can never fail (`PosInf`). It is worth mentioning that the DFT gates accept inputs that are time-to-failure functions, which allows constructing complex DFT models. The structure function of a given DFT can be expressed using the AND, OR and temporal operators. For example, the PAND-gate can be expressed as: $Y \cdot (X \trianglelefteq Y)$. Several simplification properties are introduced in [12] that allow simplifying the structure function of a given DFT in order to facilitate the analysis, such as the commutativity and idempotence properties of the OR and AND-gates. These simplification properties are formally verified using HOL4 [7], which ensures their correctness. The verification of these properties is based mainly on the definitions of the operators and the properties of `extreal` numbers. For example, `D_OR X X = X`, is verified based on the definition of the OR gate and the properties of the `extreal min` function. However, since the DFT operators of this algebra are binary operators, the simplification properties cannot support rewriting DFTs with n-ary gates. Thus, they cannot support the simplification of generic DFTs, which is the scope of the current work.

4 HOL Formalization of n-ary DFT Gates

In order to verify the DFT rewriting rules, presented in [10], we need to handle DFT gates with an arbitrary number of inputs. Therefore, we extend the definitions of DFT gates of [6]. In these definitions, we utilize lists to represent the arbitrary number of inputs. In other words, the input of an n-ary gate is a list of arbitrary size of time-to-failure functions that represent inputs of a DFT gate.

We formally define the n-ary AND-gate as:

Definition 1. ⊢ ∀L. n_AND L = FOLDR (λ a b. D_AND a b) ALWAYS L

where `FOLDR` is used to apply a binary (2-input) function over a list from right to left. The function in our case here is the binary `D_AND` that accepts two inputs and returns their result of the DFT AND operation between them. `FOLDR` requires including an element that is used to apply the function to the last element of the

input list. We use ALWAYS in this case as it is the identity element of the AND and does not affect its behavior. L represents the list of inputs to be ANDed. For example, n_AND [X; Y; Z] equals D_AND X (D_AND Y (D_AND Z ALWAYS)).

In a similar manner, we formally define the n-ary OR as:

Definition 2. ⊢ ∀L. n_OR L = FOLDR (λ a b. D_OR a b) NEVER L

D_OR is the function used with FOLDR in this definition. We use NEVER in this case as it is the identity element for the OR, i.e., NEVER will not affect the behavior of the OR-gate. It is worth mentioning that FOLDL can be used with these definitions as well, since the order of applying the OR and AND-gates does not matter if it starts from the left or from the right.

We formally define the n-ary PAND-gate as:

Definition 3. ⊢ ∀L. n_PAND L = FOLDL (λ a b. P_AND a b) ALWAYS L

This is similar to the previous definitions. However, since the PAND-gate requires that the input events fail from left to right, we use FOLDL in this case. We use ALWAYS as it does not affect the behavior of the PAND-gate, i.e., for any input X that is greater than or equal to 0, PAND ALWAYS X = X.

The VOT_k (k out of n) gate can be defined using the n_OR and n_AND gates. Firstly, we need to get the combinations that lead to the failure of the VOT-gate. For example, a (2/3) VOT-gate requires having all possible pairs out of the three inputs. Therefore, we first need to get all the possible k elements of the input list. We define k_out that accepts a list and a number k, which identifies the number of elements to be retrieved from the input list.

Definition 4. ⊢ ∀k L. k_out k L = {s| s ⊆ (set L) ∧ (CARD s = k)}

where set L returns a set with the elements in list L, and CARD is a HOL function that returns the cardinality (number of elements) of a given set. This definition basically returns a set of sets, where the inner sets are subsets of set L. This means that these inner subsets contain elements from the input list L. The added condition is that the cardinality of each of these sets equals k. As a result, we get all possible combinations of the input list that have k elements.

We use k_out to define the VOT-gate by ANDing the elements of each inner set, then ORing the result of this ANDing. We need to recall that the n_AND and n_OR accept inputs as lists not sets. Therefore, we apply a function that converts a set into a list (SET_TO_LIST). We formally define the VOT-gate as:

Definition 5. ⊢ ∀k L. k_out_n_gate k L =
n_OR (MAP (λa. n_AND (SET_TO_LIST a)) (SET_TO_LIST (k_out k L)))

where SET_TO_LIST is a HOL4 function that accepts a set and returns a list of the elements of this set. MAP is used to map a function over a list and returns a list of the mapped elements. In this definition, we first convert the outer set of k_out to a list using SET_TO_LIST (k_out k L). Then, we apply n_AND to each element of this list using MAP and convert each inner set to a list. Finally, the

n_OR is applied to the result of the MAP, i.e., the result will be the OR of ANDs and each AND has only k elements of the input list. We verify several properties for k_out and the VOT-gate, such as the finiteness of the inner and outer sets, besides other properties that are useful in the verification of the DFT rewriting rules. The HOL4 script can be accessed from [5].

5 Formal Verification of Rewriting Rules

We list the verification details of some of the rewrite rules described in Sect. 2. The details of verifying the rest of the rules can be accessed from [5].

General Rewrite Rules. The structural rewrite rules 1–5 and 7 are verified based on the definitions of n-ary gates and some list and extreal number theories properties, whereas rule 6 is implemented implicitly in the DFT formalization.

Commutativity of static gates (Rule 1)
Theorem 1. ⊢ ∀L_1 L_2. PERM L_1 L_2 ⇒ (n_AND L_1 = n_AND L_2)
Theorem 2. ⊢ ∀L_1 L_2. PERM L_1 L_2 ⇒ (n_OR L_1 = n_OR L_2)
Theorem 3. ⊢ ∀L_1 L_2 k.
PERM L_1 L_2 ⇒(k_out_n_gate k L_1 = k_out_n_gate k L_2)

The commutativity property indicates that the order of the inputs of any static gate will not affect its behavior, i.e., the time of failure for the output of the gate remains the same. We use the permutation of two lists (PERM L_1 L_2) to add the condition that L_1 and L_2 have the same inputs but with different orders. We verify the commutativity of the n_AND and n_OR gates using induction, FOLDR definition and some properties of the 2-input AND and OR-gates, defined in Sect. 3, such as associativity and commutativity. The proof of the commutativity property for the VOT-gate is mainly based on the following lemma:

Lemma 1. ⊢ ∀L_1 L_2 k. PERM L_1 L_2 ⇒ (k_out k L_1 = k_out k L_2)

which states that the sets returned by k_out are the same for two lists that have the same elements with different orders.

Gate with a single successor (Rule 3)
Theorem 4. ⊢ ∀x. rv_gt0 [x] ⇒ (n_AND [x] = x)
Theorem 5. ⊢ ∀x. n_OR [x] = x
Theorem 6. ⊢ ∀x. rv_gt0 [x] ⇒ (k_out_n_gate 1 [x] = x)
Theorem 7. ⊢ ∀x. rv_gt0 [x] ⇒ (n_PAND [x] = x)

For the static gates and the n_PAND gate, if the input list consists of only one element, then the output fails once the single input fails. The function rv_gt0 ensures that the inputs of the gates are greater than or equal to 0, which is valid as we are dealing with time-to-failure functions. We recursively define rv_gt0 as:

Definition 6. *rv_gt0*
`(rv_gt0 [] = T) ∧ (∀h t. rv_gt0 (h::t) = (∀s. 0 ≤ h s) ∧ rv_gt0 t)`

For `n_AND` and `n_OR`, rule 3 is verified based on some properties of the `D_AND` and `D_OR` gates. For VOT-gate, we use the VOT (1/n) property (Theorem 25) that replaces the VOT-gate with the `n_OR` gate. Finally, we verify rule 3 for `n_PAND` using its definition and some list and extreal numbers properties.

Left-flattening of AND-/OR-/PAND-gates (Rule 5)

Theorem 8. $\vdash \forall L_1\ L_2.$
`rv_gt0 (L1 ++ L2) ⇒ (n_AND (n_AND L2::L1) = n_AND (L2 ++ L1))`
Theorem 9. $\vdash \forall L_1\ L_2.$ `n_OR (n_OR L2::L1) = n_OR (L2 ++ L1)`
Theorem 10. $\vdash \forall L_1\ L_2.$
`rv_gt0 (L1 ++ L2) ⇒(n_PAND (n_PAND L2::L1) = n_PAND (L2 ++ L1))`

In order to verify Theorem 8, we first verify the `n_AND` append property that would split the AND of two appended lists as:

Lemma 2. $\vdash \forall L_1\ L_2.$
`rv_gt0 (L1 ++ L2) ⇒ (n_AND (L1 ++ L2) = D_AND (n_AND L1)(n_AND L2))`

where ++ is a list operator used to append two lists. We verify Theorem 8 by first rewriting `n_AND L2::L1` as `[n_AND L2]++L1`, where `::` is a list operator used to add an element to a list, which in the considered case is `n_AND L2`. Then, we use Lemma 2 to rewrite the left hand side of Theorem 8 to `D_AND (n_AND [n_AND L2])(n_AND L1)` and use Theorem 4 to verify Theorem 8. In a similar way, we verify Theorem 9 by verifying a lemma for appending two lists with `n_OR` as:

Lemma 3. $\vdash \forall L_1\ L_2.$ `n_OR (L1 ++ L2) = D_OR (n_OR L1)(n_OR L2)`

For the left-flattening property of the `n_PAND` gate, we first verify a lemma that `rv_gt0 L⇒ ∀s. 0 ≤ n_PAND L s`, which states that the output of the `n_PAND` gate is greater than or equal to 0 if the inputs follow the same condition. Theorem 10 is then verified based on the previous lemma, induction on the list argument and some `P_AND` and list properties.

Identical leftmost successors of AND, OR or PAND (Rule 7)

Theorem 11. $\vdash \forall x\ L.$ `n_AND (x::x::L) = n_AND (x::L)`
Theorem 12. $\vdash \forall x\ L.$ `n_OR (x::x::L) = n_OR (x::L)`
Theorem 13. $\vdash \forall x\ L.$ `rv_gt0 [x] ⇒ (n_PAND (x::x::L) = n_PAND (x::L))`

Theorems 11 and 12 are verified based on the definitions of `n_AND` and `n_OR` with the associativity and idempotence of `D_AND` and `D_OR` gates. Theorem 13 requires verifying that the output of a 2-input PAND-gate (`P_AND` defined in Sect. 3) with an input that already failed (`ALWAYS`) as the left input fails with the failure of the second (right) input.

Lemma 4. ⊢ ∀X. (∀s. 0 ≤ X s) ⇒ (P_AND ALWAYS X = X)

Finally, we verify the idempotence property of the P_AND gate.

Lemma 5. ⊢ ∀X. P_AND X X = X

Subsumption of OR-gates by AND-gates (Rule 8)
Theorem 14. ⊢ ∀X Y. D_AND X (D_OR X Y) = X
Subsumption of AND-gates by OR-gates (Rule 9)
Theorem 15. ⊢ ∀X Y. D_OR X (D_AND X Y) = X
Distributing OR-gates over AND-gates (Rule 10)
Theorem 16. ⊢ ∀X Y Z. D_OR (D_AND X Y)(D_AND Y Z) = D_AND (D_OR X Z) Y

We verify the rules 8–10 that are concerned with the standard axioms of Boolean algebra based on basic properties of D_AND and D_OR gates, such as the commutativity and distributivity of the AND over the OR.

OR-gates with fail-safe (NEVER) successors (Rule 11)
Theorem 17. ⊢ ∀L_1 L_2. n_OR (L_1 ++ [NEVER] ++ L_2) = n_OR (L_1 ++ L_2)
OR-gates with already failed (ALWAYS) successors (Rule 12)
Theorem 18. ⊢ ∀L_1 L_2.
rv_gt0 (L_1 ++ L_2) ⇒ (n_OR (L_1 ++ [ALWAYS] ++ L_2) = ALWAYS)

Rewrite rules 11–14 deal with scenarios that include fail-safe (NEVER) or CONST(⊥), and failed (ALWAYS) or CONST(⊤).

For Theorem 17, we use Lemma 3 and the definition of n_OR with the property stating that ∀X. D_OR X NEVER = X. We verify Theorem 18 based on Lemma 3 and the definition of n_OR along with the following lemma:

Lemma 6. ⊢ ∀X. (∀s. 0 ≤ X s) ⇒ (D_OR X ALWAYS = ALWAYS)

Then, we verify that the output of the n_OR is greater than or equal to 0 if the inputs are all greater than or equal to 0. Theorem 18 is then verified using the previous lemmas and some properties of the D_OR gate.

AND-gate with a fail-safe (NEVER) successor (Rule 13)
Theorem 19. ⊢ ∀L_1 L_2.
rv_gt0 (L_1 ++ L_2) ⇒(n_AND (L_1 ++ [NEVER] ++ L_2) = NEVER)
Theorem 20. ⊢ ∀L. rv_gt0 L ⇒ (n_PAND (L ++ [NEVER]) = NEVER)
Theorem 21. ⊢ ∀L. rv_gt0 L ⇒ (n_PAND (NEVER::L) = NEVER)
Theorem 22. ⊢ ∀L_1 L_2.
rv_gt0 (L_1 ++ L_2) ⇒(n_PAND (L_1 ++ [NEVER] ++ L_2) = NEVER)

We verify Theorem 19 using Lemma 2 and some properties for the D_AND, such as the commutativity property and ANDing with NEVER.

We verify this rule for PAND-gate by verifying two cases. Firstly, we verify that the output of the PAND cannot fail if the NEVER input is the rightmost input (Theorem 20). This is mainly verified based on some list properties to manipulate rv_gt0 along with the left flattening property of the PAND (Theorem 10). Similarly, we verify the second case when the left most input of the PAND-gate is fail-safe (Theorem 21). Finally, we verify a generic property, where the fail-safe input can be at any position (Theorem 22).

AND-gate with a failed (ALWAYS) element as successor (Rule 14)

Theorem 23. ⊢ ∀L. rv_gt0 L ⇒ (n_AND (ALWAYS::L) = n_AND L)

Theorem 24. ⊢ ∀L. rv_gt0 L ⇒ (n_PAND (ALWAYS::L) = n_PAND L)

Theorem 23 is verified using the definition of the n_AND gate with the property that the output of the gate is greater than or equal to 0 if the inputs satisfy the same condition. We verify Theorem 24 based on the definition of the n_PAND and the idempotence property of the PAND-gate.

The VOT-gate can behave as an OR-gate, when $k = 1$ (Rule 15), and as an AND-gate, when k equals the number of its inputs (Rule 16). The verification details of these rules are listed below.

Voting (1/n) is an OR-gate (Rule 15)

Theorem 25. ⊢ ∀L.
ALL_DISTINCT L ∧ rv_gt0 L ⇒ (k_out_n_gate 1 L = n_OR L)

As mentioned previously, the voting gate is defined as the OR of a list and each element in the list is the AND of another list of k elements. In order to verify Theorem 25, we need to use the commutativity property of the n_OR gate (Theorem 2), i.e, we need to verify that the list of the n_OR in the voting gate definition (MAP (λa. n_AND (SET_TO_LIST a))(MAP (λa. {a}) L)) and the input list L possess the permutation property when $k = 1$. Therefore, we first verify that the list generated from k_out 1 L is the permutation of the list MAP (λa. {a}) L. We need to recall that MAP (λa. {a}) L generates another list that has all elements from the input list L but as sets. Then, we verify that the list generated from applying the n_AND to the list of k_out 1 L is the permutation of applying n_AND to MAP (λa. {a}) L. We also verify the following property:

Lemma 7. ⊢ ∀L. rv_gt0 L ⇒
PERM (MAP (λa. n_AND (SET_TO_LIST a)) (MAP (λa. {a}) L)) L

Finally, we use these verified properties of permutation and the commutativity property of n_OR to verify Theorem 25.

Voting (n/n) is an AND-gate (Rule 16)
Theorem 26. ⊢ ∀L.
`ALL_DISTINCT L ⇒ (k_out_n_gate (LENGTH L) L = n_AND L)`

Theorem 26 is used when k equals the length of the input list (`LENGTH L`), i.e., VOT (n/n), and n is the number of inputs of the gate. In this case, the VOT-gate acts as an AND-gate. We verify this by first rewriting using the VOT-gate and `k_out` definitions. Then, we verify that {`s| s ⊆ set L ∧ (CARD s = LENGTH L)`} = {`set L`}. This way the original expression of the VOT-gate can be reduced to `n_OR [n_AND (SET_TO_LIST (set L))]`. Then, we verify that `PERM L (SET_TO_LIST (set L))`, which means that the original list and the list generated from the set of the original list are the permutation of each other. This is a consequence of using `set L` in the formal definition of the VOT-gate, which requires the added condition that the elements in the original list are distinct, i.e., they are not equal or repeated. This condition is added using the HOL predicate `ALL_DISTINCT L`. Finally, we verify Theorem 26 using the commutativity property of the AND (Theorem 1) and the definition of `n_OR`.

Rewrite Rules for PAND-gates. Rules 18–23 deal with PAND-gates that require considering the order of the inputs.

Representing AND-gate using OR- and PAND-gates (Rule 18)
Theorem 27. ⊢ `∀X Y. D_AND X Y = D_OR (P_AND X Y) (P_AND Y X)`
Conflicting PAND-gates with independent successors (Rule 19)
Theorem 28. ⊢ `∀X Y.`
`(∀s. ALL_DISTINCT [X s; Y s]) ⇒ (D_AND (P_AND X Y) (P_AND Y X) = NEVER)`

We verify Theorems 27 and 28 based on the definitions of `D_AND`, `D_OR` and `P_AND` gates and some properties of the extreal numbers. Note that the added condition for rule 19 is that the inputs are distinct (`ALL_DISTINCT`), i.e., they cannot fail simultaneously. This results from the fact that the inputs are independent (there is no common cause of failure) and they possess continuous failure distributions. Therefore, rule 19 cannot be applied unless this context restriction is ensured using this assumption.

PAND-gate with a PAND-successor (Rule 20)
Theorem 29. ⊢ `∀B C1 C2 L. rv_gt0 (L ++ [B; C1; C2])⇒`
`(n_PAND ([B; P_AND C1 C2] ++ L) =`
`D_AND (P_AND C1 C2) (n_PAND ([B; C2] ++ L)))`

We verify Theorem 29 based on manipulating the input lists and the PAND appended with a single element lemma, which we verify as:

Lemma 8. ⊢ ∀x L. rv_gt0 L ⇒ (n_PAND (L ++ [x]) = P_AND (n_PAND L) x)

Based on Lemma 8 and list induction and manipulation, we verify that the left-hand-side of Theorem 29 equals: P_AND(D_AND(P_AND C1 C2)(n_PAND (B::C2::L))) x, where x is the additional element generated through induction. Then, we verify a property stating that the time of failure of the PAND-gate should be greater than or equal to the failure time of any of its inputs, since it is required that the failure to occur from left to right.

PAND-gate with a first OR-successor (Rule 21)

Theorem 30. ⊢ ∀X Y L. rv_gt0 [X; Y] ⇒
(n_PAND (D_OR X Y::L) = D_OR (n_PAND (X::L)) (n_PAND (Y::L))

To verify Theorem 30, we first apply induction to the input argument and rewrite using the rule of n_PAND with a single successor. Then, we use the definitions of the P_AND, n_PAND and some simplification theorems, such as P_AND ALWAYS X = X. Using some list properties, such as applying a function to two appended list using FOLDL (we need to recall that the definition of n_PAND is based on FOLDL), we reach a point where the whole goal is similar to and can be verified using the following lemma:

Lemma 9. ⊢ ∀X Y Z. P_AND (D_OR X Y) Z = D_OR (P_AND X Z)(P_AND Y Z)

PAND-gate with ALWAYS as non-first successor (Rule 23)

Theorem 31. ⊢ ∀L_1. $L_1 \neq$ [] ∧ (∀x. MEM x L_1 ⇒ ∀s. 0 < x s) ⇒
∀ L_2. n_PAND (L_1 ++ [ALWAYS] ++ L_2) = NEVER

Theorem 31 shows that if the inputs to the left of the input that already failed (ALWAYS) do not fail from the beginning, i.e., their time of failure is greater than 0, then the output of the n_PAND can never fail. Therefore, we add the condition that the inputs to the left (list L_1) are greater than 0 using ∀x. MEM x L_1 ⇒ ∀s. 0 < x s. We verify Theorem 31 using induction over list L_1. After some basic list and extreal theory based reasoning, we reach the step for the left-hand-side:

FOLDL (λa b. P_AND a b)
(P_AND (FOLDL(λa b. P_AND a b) h L_1) ALWAYS) L_2

where h is the appended element that results from induction. We verify that P_AND (FOLDL(λa b. P_AND a b) h L_1) ALWAYS = NEVER, which can be done if the first input of the P_AND is greater than 0. We verify the following property:

Lemma 10. ⊢ ∀s L. (∀x. MEM x L ⇒ ∀s. 0 < x s) ⇒
∀h. 0 < h s ⇒ 0 < FOLDL (λa b. P_AND a b) h L s

This lemma basically means that if we have a list of inputs and an additional element, h, that are greater than 0, then the result of applying `P_AND` using `FOLDL` is also greater than 0. Using this lemma, the left hand side is reduced to `FOLDL (λa b. P_AND a b) NEVER` L_2. Finally, we use the following lemma to verify the Theorem 31.

Lemma 11. $\vdash \forall$`L. FOLDL (λa b. P_AND a b) NEVER L = NEVER`

This lemma indicates that if we apply `P_AND` to a list of inputs with an element `NEVER` at the beginning, then the output equals `NEVER`

Non-structural Rules. The BEs that are not connected to the given DFT can be safely removed. This is already implicitly embedded in the current DFT formalization, as we are verifying the rewrite rules by proving that the time of failure before and after rewriting remains the same. Therefore, if the BEs are not connected to the DFT, this means that they are not affecting the time of failure of the top element and thus they can be removed in the verification process. Since DFT gates are modeled as time-to-failure functions, merging BEs is also already embedded in the DFT formalization. For example, the OR-gate is modeled using the `min` function. This means that the inputs of the OR-gate are merged and the output of the OR-gate can be replaced with the `min` function.

We illustrate the usage of the verified rules on the example of Fig. 3:

Theorem 32. $\vdash \forall$`.c d f`
`P_AND (D_AND c (D_OR c d))(D_AND d f) = P_AND c (D_AND d f)`

In this section, we presented the formal definitions and proofs of the rewriting rules in [10], which we believe is a novel contribution as details about how to mathematically conduct these proofs are not available in [10]. In fact, in [10], the correctness of the rewrite rules is described inexplicitly based on the behavior of DFT gates rather than their formal mathematical models as presented in this paper. It is worth noting that our formal definitions and verified lemmas allowed verifying several DFT rewriting rules that can be used with tools that simplify DFTs prior to the analysis. In addition, verifying these rules represent the first step towards formally verifying tools, such as Storm, that support DFT analysis and use these rewriting rules. The HOL4 script for these rules and their lemmas is comprised of about 1500 lines and required about 80 h to develop. The script is available at [5].

6 Conclusions

In this paper, we provided the formal verification of DFT rewriting rules using the HOL4 theorem prover. These rules enable simplifying DFTs before performing the analysis through tools, such as the Storm model checker. In order to verify the rules, we formally defined n-ary gates, such as AND, OR, PAND and VOT-gates and verified several lemmas based on these definitions and the

available DFT theory in HOL4. We mainly verified DFT rules that deal with the static gates (AND, OR & VOT-gates) and the PAND-gate. The rules include some known properties, such as the commutativity of the static gates. Moreover, we verified some more complex rules that deal with PAND with different input scenarios. The formal verification of the rewriting rules in the DFTs analysis adds the confidence level of the results of the tools that use them. We plan to extend this work to verify rewriting rules that include Functional DEPendency (FDEP) and Spare gates as well. This work can be considered as a first milestone for formally verifying automated DFT analysis tools such as Storm.

Acknowledgments. The authors would like to thank Sebastian Junges, from RWTH Aachen University, for the discussions and comments on the rewrite rules.

References

1. Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic fault tree analysis using input/output interactive Markov chains. In: Proceedings of DSN, pp. 708–717. IEEE (2007)
2. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
3. Dugan, J.B., Bavuso, S.J., Boyd, M.A.: Fault trees and sequence dependencies. In: Proceedings of RAMS, pp. 286–293 (1990)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31188-2
5. Elderhalli, Y.: DFT rewriting rules: HOL4 script, Concordia University, Montreal, QC, Canada (2019). http://hvg.ece.concordia.ca/code/hol/DFT-rewrite/index.php
6. Elderhalli, Y., Ahmad, W., Hasan, O., Tahar, S.: Probabilistic analysis of dynamic fault trees using HOL theorem proving. J. Appl. Log. **6**, 467–509 (2019)
7. Elderhalli, Y., Hasan, O., Ahmad, W., Tahar, S.: Formal dynamic fault trees analysis using an integration of theorem proving and model checking. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 139–156. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_10
8. Ghadhab, M., Junges, S., Katoen, J., Kuntz, M., Volk, M.: Safety analysis for vehicle guidance systems with dynamic fault trees. Reliab. Eng. Syst. Saf. **186**, 37–50 (2019)
9. HOL4 (2019). https://hol-theorem-prover.org/
10. Junges, S., Guck, D., Katoen, J., Rensink, A., Stoelinga, M.: Fault trees on a diet: automated reduction by graph rewriting. Form. Asp. Comput. **29**(4), 651–703 (2017)
11. Junges, S.: Simplifying dynamic fault trees by graph rewriting. Master thesis, RWTH Aachen University (2015)
12. Merle, G.: Algebraic modelling of dynamic fault trees, contribution to qualitative and quantitative analysis. Ph.D. thesis, ENS Cachan, France (2010)

13. Mhamdi, T., Hasan, O., Tahar, S.: On the formalization of the lebesgue integration theory in HOL. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 387–402. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_27
14. Mhamdi, T., Hasan, O., Tahar, S.: Formalization of entropy measures in HOL. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 233–248. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22863-6_18
15. Ruijters, E., Stoelinga, M.: Fault tree analysis: a survey of the state-of-the-art in modeling, analysis and tools. Comput. Sci. Rev. **15–16**, 29–62 (2015)
16. Stamatelatos, M., Vesely, W., Dugan, J., Fragola, J., Minarick, J., Railsback, J.: Fault Tree Handbook with Aerospace Applications. NASA Office of Safety and Mission Assurance (2002)
17. Volk, M., Junges, S., Katoen, J.P.: Fast dynamic fault tree analysis by model checking techniques. IEEE Trans. Ind. Inform. **14**(1), 370–379 (2018)