**Technical Report soft-11-02, Chair for Software Engineering, University of Konstanz**

# From Probabilistic Counterexamples via Causality to Fault Trees

Matthias Kuntz[1], Florian Leitner-Fischer[2], and Stefan Leue[2]

[1] TRW Automotive GmbH, Germany
[2] University of Konstanz, Germany

**Abstract.** Probabilistic Model Checking is an established technique used in the dependability analysis of safety-critical systems. In recent years, several approaches to generate probabilistic counterexamples have been proposed. The interpretation of stochastic counterexamples, however, continues to be problematic since they have to be represented as sets of paths, and the number of paths in this set may be very large. Fault trees (FTs) are a well-established industrial technique to represent causalities for possible system hazards resulting from system or system component failures. In this paper we suggest a method to automatically derive FTs from counterexamples, including a mapping of the probability information onto the FT. We extend the structural equation approach by Pearl and Halpern, which is based on Lewis counterfactuals, so that it serves as a justification for the causality that our proposed FT derivation rules imply. The synthesized FTs provide the user with a concise and compact representation of the causes of potential system failures, together with their respective probabilities. We demonstrate the usefulness of our approach by applying it to a selection of industrial size case studies.

## 1 Introduction

In recent joint work [2] with our industrial partner TRW Automotive GmbH we have proven the applicability of stochastic formal analysis techniques to safety analysis in an industrial setting. The analysis approach that we used was that of probabilistic failure modes effect analysis (pFMEA) [19]. In [2] we showed that counterexamples are a very helpful means to understand how certain error states representing hazards can be reached by the system. While the visualization of the graph structure of a stochastic counterexample [3] helps engineers to analyze the generated counterexample, it is still difficult to compare the thousands of paths in the counterexample with each other, and to discern causal factors during fault analysis.

In safety analysis, fault tree analysis (FTA) [34] is a well-established industrial method to break down the hazards occurring in complex, technical systems into a combination of what is referred to as basic events, which represent system

component failures. It uses a widely accepted graphical notation to represent the resulting fault trees. The main drawback of fault tree analysis is that it relies on the ability of the safety engineer to manually identify all possible component failures that might cause a certain hazard. In this paper we present a method that automatically generates a fault tree from a probabilistic counterexample. Our method provides a compact and concise representation of the system failures using a graphical notation that is well known to safety engineers. At the same time the derived fault tree constitutes an abstraction of the probabilistic counterexample since it focuses on representing the implied causalities rather than enumerating all possible execution sequences leading to a hazard. The causality expressed by the fault tree is rooted in the counterfactual notion of causality that is widely accepted in the literature.

Our approach can be described by identifying the following steps:

- Our fault tree computation method uses a system model given in the input language of the PRISM probabilistic model checker [22].
- For this model we compute counterexamples for stochastic properties of interest, representing system hazards, using our counterexample computation extension of PRISM called DiPro [3]. The counterexamples consist of potentially large numbers of system execution paths and their related probability mass information.
- In order to compute fault trees from these counterexamples we compute what is commonly referred to as basic events. Those are events that cause a certain hazard. The fault tree derivation is implemented in a tool called CX2FT.
- The justification for the fault tree computation is derived from a model of causality due to Halpern and Pearl [20] that we modify and extend to be applicable to our setting.
- The path probabilities computed by the stochastic model checker are then mapped on the computed fault tree.
- Finally, the obtained fault tree is represented graphically by an adapted version of the FaultCAT tool[3].

All analysis steps are fully automated and do not require user intervention. We demonstrate the usefulness of our approach by applying it to a selection of case studies known from the literature on stochastic model checking. In Fig. 1, a pictorial overview of our approach can be found.

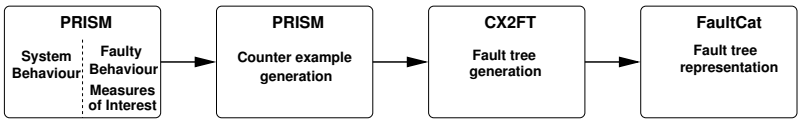


**Fig. 1.** Tool Chain for Generating Fault Trees out of Counterexamples

[3] http://www.iu.hio.no/FaultCat/

This paper is organized as follows: In Section 2 we briefly introduce the concepts of counterexamples in stochastic model checking and fault trees. In Section 3 we describe the model of causality that we use, and how probabilistic counterexamples can be mapped to fault trees. In Section 4 we demonstrate the application of our approach to case studies known from the literature. A discussion of related work follows in Section 5. Finally, Section 6 concludes the paper with a summary and an outlook on future research.

## 2 Counterexamples and Fault Trees

### 2.1 Counterexamples in Stochastic Model Checking

In stochastic model checking, the property that is to be verified is specified using a variant of temporal logic. The temporal logic used in this paper is Continuous Stochastic Logic (CSL) [1, 7]. It is tailored to reason about quantitative system behavior, including the performance and dependability of a system. Just like in traditional model checking, given an appropriate system model and a CSL property, stochastic model checking tools such as PRISM [22] or MRMC [23] can verify automatically whether the model satisfies the property. If the model refutes the property, a counterexample usually helps engineers to comprehend the reasons of the property violation and to correct the error. Stochastic model checkers do not automatically provide counterexamples, but the computation of counterexamples has recently been addressed in [5, 4, 6, 17, 21, 33].

For the purpose of this paper it suffices to consider only upper bounded probabilistic timed reachability properties. They require that the probability of reaching a certain state, often corresponding to an undesired system state, does not exceed a certain upper probability bound $p$. In CSL such properties can be expressed by formulae of the form $\mathcal{P}_{\leq p}(\varphi)$, where $\varphi$ is path formula specifying undesired behavior of the the system. Any path which starts at the initial state of the system and which satisfies $\varphi$ is called a *diagnostic path*. A counterexample for an upper bounded property is a set $X$ of diagnostic paths such that the accumulated probability of $X$ violates the probability constraint $\leq p$. If the CSL formula $\mathcal{P}_{=?}\ (\varphi)$ is used, the probability of the path formula $\varphi$ to hold is computed and the counterexample contains all paths fulfilling $\varphi$. The probability of the counterexample is computed using a stochastic model checker, in our case PRISM. Notice that in the setting of this paper the counterexample is computed completely, i.e., all simple paths leading into the undesired system state are enumerated in the counterexample.

### 2.2 Fault Trees and Fault Tree Analysis

Fault trees (FTs) [34] are being used extensively in industrial practice, in particular in fault prediction and analysis, to illustrate graphically under which conditions systems can fail, or have failed. In our context, we need the following elements of FTs:

1. Basic event: represents an atomic event.
2. *AND*-gate: represents a failure, if all of its input elements fail.
3. *OR*-gate: represents a failure, if at least one of its input elements fails.
4. Priority-AND (*PAND*): represents a failure, if all of its input elements fail in the specified order. The required input failure order is usually read from left to right.
5. Intermediate Event: failure events that are caused by their child nodes. The probability of the intermediate event to occur is denoted by the number in the lower right corner. A top level event (TLE) is a special case of an intermediate event, representing the system hazard.

The graphical representation of these elements can be found in Fig. 2. The *AND*-, *OR*- and *PAND*-gates are used to express that their top events are caused by their input events. For an in-depth discussion of fault trees we refer the reader to [34].

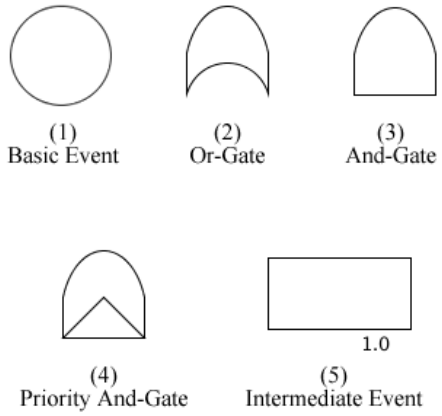


**Fig. 2.** Fault Tree Elements

*Example 1.* In Figure 3 we present an FT describing the conditions under which a simple 2-out-of-3 fault-tolerant, redundant system is failed. To be failed, components $A$ and $B$ or $A$ and $C$ or $B$ and $C$, or $A$, $B$ and $C$ have to be failed. FTs without $PAND$ can be rewritten as Boolean formulae. The FT of the 2-out-of-3 fault tolerant system can be represented the following boolean formula:

$$(A \wedge B) \vee (A \wedge C) \vee (B \wedge C) \vee (A \wedge B \wedge C).$$

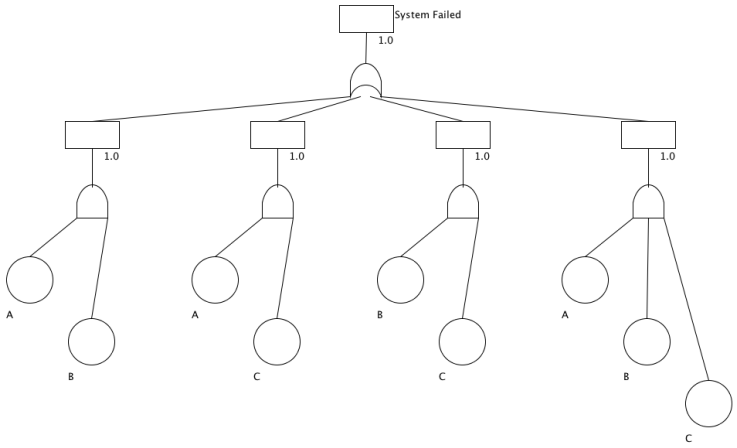


**Fig. 3.** Fault Tree Representation of a 2-out-of-3-System Failure

## 3 Computing Fault Trees from Counterexamples

### 3.1 Inferring Causality

Fault Trees express causality, in particular they characterize basic events as being causal factors in the occurrence of the top-level event in some Fault Tree. The counterexamples that we use to synthesize these causal relationships, however, merely represent possible executions of the system model, and not explicitly causality amongst event occurrences. The system models that we consider correspond to concurrent system executions, and hence define partial order relations on the events in the system [24]. Each path in the counterexample is a linearly ordered, interleaved sequence of events consistent with this partial order. The question is hence how, and with which justification, we can infer causality from the sets of linearly ordered event sequences that we obtain in the course of the counterexample computation.

We use the concept of *structural equations* as proposed by Halpern and Pearl [20] as a model of causality. It is based on *counterfactual* reasoning and the related *alternative world* semantics of Lewis [26, 14, 28]. The counterfactual argument is widely used as the foundation for identifying faults in program debugging [35, 18] and also underlies the formal fault tree semantics proposed in [32]. The "naive" counterfactual causality criterion according to Lewis is as follows: event $A$ is causal for the occurrence of event $B$ if and only if, were $A$ not to happen, $B$ would not occur. The testing of this condition hinges upon the availability of alternative worlds. A causality can be inferred if there is a world in which $A$ and $B$ occur, whereas in an alternative world neither $A$ nor $B$ occurs.

The naive interpretation of the Lewis counterfactual test, however, leads to a number of inadequate or even fallacious inferences of causes, in particular

if causes are given by combinations of multiple events. The problematic issues include common or hidden causes, the disjunction and conjunction of causal events, the non-occurrence of events, and the preemption of failure causes due to, e.g., repair mechanisms. A detailed discussion of these issues is beyond the scope of this paper, and we refer to the critical literature on counterfactual reasoning, e.g., [14, 28].

Since we are considering concurrent systems in which particular event interleavings may be the cause of errors, e.g., race conditions, the order of occurrence of events is an potential causal factor that cannot be disregarded. Consider a railroad crossing model in which $G$ denotes the gate closing, $O$ the gate opening, $T$ the train crossing the road, and $C$ the car crossing the tracks. A naive counterfactual test will fail to show that the event sequence $< G, O, T, C >$ is a potential cause of a hazard, whereas $< G, T, O, C >$ is not. In addition, the naive counterfactual test may determine irrelevant causal events. For instance, the fact that the train engineer union has decided not to call for a strike is not to be considered a cause for the occurrence of an accident at the railroad crossing.

Halpern and Pearl extend the Lewis counterfactual model in [20] to what they refer to as *structural equation model* (SEM). It encompasses the notion of causes being logical combinations of events as well as a distinction of relevant and irrelevant causes. However, the structural equation model does not account for event orderings, which is a major concern of this paper.

We now sketch an actual cause definition adopted from [20]. An actual cause is a cause in which irrelevant events are factored out. A causal formula in the SEM is a boolean conjunction $\psi$ of variables representing the occurrence of events. We only consider boolean variables, and the variable associated with an event is true in case that event has occurred. The set of all variables is partitioned into the set $U$ of *exogenous* variables and the set $V$ of *endogenous* variables. Exogenous variables represent facts that we do not consider to be causal factors for the effect that we analyze, even though we need to have a formal representation for them so as to encode the "context" ([20]) in which we perform causal analysis. An example for an exogenous variable is the train engineer union's decision in the above railroad crossing example. Endogenous variables represent all events that we consider to have a meaningful, potentially causal effect. The set $X \subseteq V$ contains all events that we expect jointly to be a candidate cause, and the boolean conjunction of these variables forms a causal formula $\psi$. The causal process comprises all variables that mediate between $X$ and the effect that $\psi$ is causing. Those variables are not root causes, but they contribute to rippling the causal effect through the system until reaching the final effect. Omitting a complete formalization, we assume that there is an actual world and an alternate world. In the actual world, there is a function $val_1$ that assigns values to variables. In the alternate world, there is a function $val_2$ assigning potentially different values to the variables.

In the SEM, a formula $\psi$ is an actual cause for an event represented by the formula $\varphi$, if the following conditions are met:

AC1: Both $\psi$ and $\varphi$ are true in the actual world, assuming the context defined by the variables in $U$, and given a valuation $val_1(V)$.

AC2: The set of endogenous events $V$ is partitioned into sets $Z$ and $W$, where the events in $Z$ are involved in the causal process and the events in $W$ are not involved in the causal process. It is assumed that $X \subseteq Z$ and that there are valuations $val_2(X)$ and $val_2(W)$ assigning values to the variables in $X$ and $W$, respectively, such that:

1. Changing the values of the variables in $X$ and $W$ from $val_1$ to $val_2$ changes $\varphi$ from true to false.
2. Setting the values of the variables in $W$ from $val_1$ to $val_2$ should have no effect on $\varphi$ as long as the values of the variables in $X$ are kept at the values defined by $val_1$, even if all the variables in an arbitrary subset of $Z \setminus X$ are set to their value according to $val_1$.

AC3: The set of variables $X$ is minimal: no subset of $X$ satisfies conditions AC1 and AC2.

AC2(1) corresponds to the Lewis counterfactual test. However, as [20] argue, AC2(1) is too permissive, and AC2(2) constrains what is admitted as cause by AC2(1). Minimality in AC3 ensures that only those elements of the conjunction that are essential for changing $\varphi$ in AC2(1) are considered part of the cause; inessential elements are pruned.

### 3.2 Formal Representation of Events and their Order

To logically reason about the causality of events in our setting we need to allow for the description of conjunctive and disjunctive occurrence of events and represent, at the same time, the order in which the events occur.

In the common description of the structural equation model the occurrence of events is encoded as boolean formulae. In these formulae, boolean variables represent the occurrence of an event (true = event occurred, false = event did not occur). These variables are connected via the boolean and- or or-operators to express conjunctive or disjunctive constraints on their occurrence. Note that this representation does not yet allow for expressing logical constraints on the order in which events need to occur.

We first define a mathematical model that allows us to logically reason about the occurrence of events in sets of execution sequences forming counterexamples in stochastic model checking. Technical Systems evolve in discrete computation steps. A system state $s$ is defining a valuation of the system state variables. In our setting, we limit ourselves to considering systems that only contain Boolean state variables representing the occurrence of events, as described above. We use a set of atomic propositions that represent the Boolean state variables we consider. A computation step is characterized by an instantaneous transition which takes the system from some state $s$ to a successor state $s'$. The transition from $s$ to $s'$ will be triggered by an action $a$, corresponding to the occurrence of an event. Since we wish to derive causality information from sets of finite computations, which we obtain by observing a finite number of computation

steps, our main interest will be in sets of state-action sequences. We define the following model as a basis for our later formalization of the logical connection between events.

**Definition 1.** *State-Action Trace Model. Let $S$ denote a set of* states*, $AP$ a finite set of atomic state propositions, and $Act$ a finite set of action names.*

- *A finite sequence $s_0, a_0, s_1, a_1, \ldots a_{n-1}, s_n$ with, for all $i$, $s_i \in S$ and $a_i \in Act$, is called a* state-action trace *over $(S, Act)$.*
- *A* State-Action Trace Model (SATM) *is a tuple $M = (S, Act, AP, L, \Sigma)$ where $\Sigma = \{\sigma_1, \ldots \sigma_k\}$ such that each $\sigma_i$ is a state-action trace over $(S, Act)$, and $L : S \to 2^{AP}$ is a function assigning each state the set of atomic propositions that are true in that state.*

We assume that for a given SATM $M$, $AP$ contains the variables representing the events that we wish to reason about. We also assume that for a given state-action trace $\sigma$, $L(s_i)$ contains the event variable corresponding to the action $a_{i-1}$. Notice that we consider event instances, not types. In other words, the $n-th$ occurrence of some event of type $E$ will be distinct in $AP$ from the $n+1st$ occurrence of this event type.

We next define an event order logic allowing us to reason about boolean conditions on the occurrence of events. The logic is using a set $\mathcal{A}$ of event variables as well as the boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events we introduce the ordered conjuction operator $\barwedge$. The formula $A \barwedge B$ is satisfied if and only if events $A$ and $B$ occur in a trace and $A$ occurs before $B$. The formal semantics of this logic is defined on SATMs. Notice that the $\barwedge$ operator is a temporal logic operator and that the SATM model is akin to a linearly ordered Kripke structure.

**Definition 2.** *Semantics of event order logic. Let $M = (S, Act, AP, L, \Sigma)$ a SATM, $\phi$ and $\psi$ formulae of the event order logic, and let $\mathcal{A}$ a set of event variables, with $A \in \mathcal{A}$, over which $\phi$ and $\psi$ are built. Let $\sigma = s_0, a_0, s_1, a_1, \ldots a_{n-1}, s_n$ a state-action trace over $(S, Act)$. We define that a formula is satisfied in state $s_i$ of $\sigma$ as follows:*

- $s_i \vDash A$ iff $A \in L(s_i)$.
- $s_i \vDash \neg\phi$ iff not $s_i \vDash \phi$.
- $s_i \vDash \phi \wedge \psi$ iff $\exists j, k : i \leq j, k \leq n \;.\; s_j \vDash \phi$ and $s_k \vDash \psi$.
- $s_i \vDash \phi \vee \psi$ iff $\exists j, k : i \leq j, k \leq n \;.\; s_j \vDash \phi$ or $s_k \vDash \psi$.
- $s_i \vDash \phi \barwedge \psi$ iff $\exists j, k : i \leq j \leq k \leq n \;.\; s_j \vDash \phi$ and $s_k \vDash \psi$.

*We define that a sequence $\sigma$ satisfies a formula $\phi$, written as $\sigma \vDash \phi$ iff $\exists i \;.\; s_i \vDash \phi$. We define that the SATM $M$ satisfies the formula $\phi$, written as $M \vDash \phi$, iff $\exists \sigma \in \Sigma \;.\; \sigma \vDash \phi$.*

In order to perform comparison operations between paths we define a number of path comparison operators.

**Definition 3.** *Path Comparison Operators. Let* $M = (S, Act, AP, L, \Sigma)$ *a SATM, and* $\sigma_1$ *and* $\sigma_2$ *state-action traces in M.*

- $=$*:* $\sigma_1 = \sigma_2$ *iff* $\forall e \in Act \,.\, \sigma_1 \vDash e \equiv \sigma_2 \vDash e$.
- $\doteq$*:* $\sigma_1 \doteq \sigma_2$ *iff* $\forall e_1, e_2 \in Act \,.\, \sigma_1 \vDash e_1 \wedge e_2 \equiv \sigma_2 \vDash e_1 \wedge e_2$.
- $\subseteq$*:* $\sigma_1 \subseteq \sigma_2$ *iff* $\forall e \in Act \,.\, \sigma_1 \vDash e \Rightarrow \sigma_2 \vDash e$. *Furthermore,* $\sigma_1 \subset \sigma_2$ *iff* $\sigma_1 \subseteq \sigma_2$ *and not* $\sigma_1 = \sigma_2$.
- $\dot{\subseteq}$*:* $\sigma_1 \dot{\subseteq} \sigma_2$ *iff* $\forall e_1, e_2 \in Act \,.\, \sigma_1 \vDash e_1 \wedge e_2 \Rightarrow \sigma_2 \vDash e_1 \wedge e_2$. *Furthermore,* $\sigma_1 \dot{\subset} \sigma_2$ *iff* $\sigma_1 \dot{\subseteq} \sigma_2$ *and not* $\sigma_1 \doteq \sigma_2$.

We are now ready to adopt the SEM to event orders. We interpret the SEM equations over a given SATM $M$. Again, without providing a detailed formalization, we assume the existence of a function $order_1$ assigning an order to the occurrence of the events $M$ in the actual world, as well as a function $order_2$ which assigns a potentially different order in the alternate world. An event order logic formula $\psi$ is considered a cause for an event represented by the event order logic formula $\varphi$, if the following conditions are satisfied:

AC1: Both $\psi$ and $\varphi$ are true in the actual world, assuming the context defined by the variables in $U$, given a valuation $val_1(V)$ and an order $order_1(V)$.

AC2: The set of endogenous events $V$ is partitioned into sets $Z$ and $W$, where the events in $Z$ are involved in the causal process and the events in $W$ are not involved in the causal process. It is assumed that $X \subseteq Z$ and that there exist valuations $val_2(X)$ and $val_2(W)$ and orders $order_2(X)$ and $order_2(W)$ such that:

1. Changing the values of the variables in $X$ and $W$ from $val_1$ to $val_2$ and the order of the variables in $X$ and $W$ from $order_1$ to $order_2$ changes $\varphi$ from true to false.
2. Setting the values of the variables in $W$ from $val_1$ to $val_2$ and the order of the variables in $W$ from $order_1$ to $order_2$ should have no effect on $\varphi$ as long as the values of the variables in $X$ are kept at the values defined by $val_1$, and the order as defined by $order_1$, even if all the variables in an arbitrary subset of $Z \setminus X$ are set to their value according to $val_1$ and $order_1$.

AC3: The set of variables $X$ is minimal: no subset of $X$ satisfies conditions AC1 and AC2.

If a formula $\psi$ meets the above described conditions, the occurrence of the events included in $\psi$ is causal for $\varphi$. However, condition AC2 does not imply that the order of the occurring events is causal. We introduce the following condition to express that the order of the variables occurring in $\psi$, or an arbitrary subset of these variables, has an influence on the causality of $\varphi$:

OC1: Let $Y \subseteq X$. Changing the order $order_1(Y)$ of the variables in $Y$ to an arbitrary order $order_2(Y)$, while keeping the variables in $X \setminus Y$ at $order_1$, changes $\varphi$ from true to false.

If for a subset of $X$ OC1 is not satisfied, the order of the events in this subset has no influence on the causality of $\varphi$.

### 3.3 Fault Tree Generation

In order to automatically synthesize a fault tree from a stochastic counterexample, the combinations of basic events causing the top level event in the fault tree have to be identified. Recall that the top level event represents the system hazard that we wish to analyze in order to determine its causal factors. Using a stochastic model checker we compute a counterexample which contains all paths leading to a state corresponding to the occurrence of some top level event $T$. This is achieved by computing the counterexample for the CSL formula $P=?(true\ U\ t)$, where $t$ is a state formula representing the top level event $T$. We interpret counterexamples in the context of an SATM $M = (S, Act, AP, L, \Sigma)$. We assume that $\Sigma$ is partitioned in disjoint sets $\Sigma_G$ and $\Sigma_C$, where $\Sigma_C$ contains all traces belonging to the counterexample, whereas $\Sigma_G$ contains all maximal simple system traces that do not belong to the counterexample. The disjointness of $\Sigma_C$ and $\Sigma_G$ implies that $M$ is deterministic with respect to the causality of $T$. Furthermore, we define $M_C = (S, Act, AP, L, \Sigma_C)$ as the restriction of $M$ to only the counterexample traces, and refer to it as a counterexample model. W.l.o.g. we assume that every trace $\sigma \in M_C$ contains a last transition executing the top level event $T$, so that its last state $s_n \vDash T$, which implies that $M_c \vDash T$. In our interpretation of the SEM, actual world models will be derived from $\Sigma_C$, whereas alternate world models are part of $\Sigma_G$. Notice that in order to compute the full model probability of reaching $T$ it is necessary to perfom a a complete state space exploration of the model that we analyze. We hence obtain $M_G$ at no additional cost.

We next define the candidate set of paths that we consider to be causal for $T$. We define this set in such a way that it includes all minimal paths. Paths are minimal if they do not contain a subpath according to the $\subseteq$ operator that is also a member of the candidate set.

**Definition 4 (Candidate Set).** *Let $M_C = (S, Act, AP, L, \Sigma_C)$ a counterexample model, and $T$ a top level event in $M_C$. We define the candidate set of paths belonging to the fault tree of $T$ as CFT(T):*

$$CFT(T) = \{\sigma \in \Sigma_C | \forall \sigma' \in \Sigma_C \,.\, \sigma' \subseteq \sigma \Rightarrow \sigma' = \sigma\}. \tag{1}$$

Notice that the candidate set is minimal in the sense that removing an event from some path in the candidate set means that the resulting path is no longer in the counterexample $\Sigma_C$.

Given a counterexample model $M_C$, we state the following observations regarding the paths included in $\Sigma_C$:

- Each $\sigma \in \Sigma_C$ can be viewed as an ordered conjunction $A_1 \wedge \ldots \wedge A_{n-1} \wedge T$ of events, where T is the top level event that we consider.
- On each path in the counterexample, there has to be at least one event causing the top level event. If that was not the case, the top level event would not have occurred on that path and as a result the path would not be in the counterexample.

The algorithm that we propose to compute fault trees is an over-approximation of the computation of the causal events $X$ since computing the set $X$ precisely is not efficiently possible [16]. Instead of computing the set $X$ of events that are causal for some $\varphi$, we compute the set $Z$ of events, which consists of all events that are part of the causal process of $\varphi$. $Z$ will then be represented by $\psi$. Since $X$ is a subset of $Z$ we can assure that no event that is causal is omitted from the fault tree. It is, however, possible that due to our overapproximation events that are not in $X$ are added to the fault tree. This applies in particular to those events that are part of the causal process, and hence mediate between $X$ and $\varphi$. However, as we show in Section 4, adding such events can be helpful to illustrate how the root cause is indirectly propagating by non-causal events to finally cause the top level event.

We do not account for exogenous variables, since we believe them to be less relevant in the analysis of models of computational systems since the facts represented in those models all seem to be endogenous facts of the system. However, should one wish to consider exogenous variables, those can easily be retrofitted.

We now define tests that will identify the set Z of ordered conjunctions of events that satisfy the conditions AC1 to AC3 and OC1, and which hence can be viewed as part of the causal process of the top level event. The starting point for this computation is the candidate set of Definition 4.

*Test for AC1:* The actual causal set $Z$ that our algorithm computes is a subset of the events included in the candidate set $CFT(T)$ for some given top level event $T$. Since we assume that every path includes at least one causal event, $Z$ is not empty. We may hence conclude that $CFT(T) \vDash \psi$ and $CFT(T) \vDash \varphi$.

*Test for AC2(1):* We check for each path $\sigma \in$ *CFT(T)* whether the ordered conjunctions of events that it is representing fulfills the condition AC2(1). We assume that the set of events $Z$ is equal to the events occurring on the path $\sigma$. We further assume that $W = V \setminus Z$ and that $V = Act$. $W$ hence contains all events that are possible, minus the ones that occur on path $\sigma$. More formally, for a given $\sigma$, $Z = \{e \in V | \sigma \vDash e\}$. This corresponds to using the valuation $val_1$ to assign true to all variables in $Z$ and false to all variables in $W$ in the formulation of AC2(1). Changing the valuation of the variables in $Z$ to move from $val_1$ to some $val_2$ can imply removing variables from $Z$. Due to the minimality of $\sigma$ this implies that the resulting trace $\sigma'$ is no longer in $\Sigma_C$. Testing of this condition is hence implicit and implied by the minimality of the candidate set.

*Test for AC2(2):* We need to check that moving $W$ from $val_1$ to $val_2$ and from $order_1$ to $order_2$ has no effect on the outcome of $\varphi$ as long as the values of $X$ are kept at the values defined by $val_1$ and the order defined by $order_1$. Recall that $W$ denotes all events that are not currently considered to be causal, and that we compute $Z$ as an overapproximation of $X$. For a path $\sigma \in$ *CFT(T)* changing $W$ from $val_1$ to $val_2$ and from $order_1$ to $order_2$ implies that events are added to $\sigma$. Thus, we check for each path $\sigma \in$ *CFT(T)* whether there exists some path $\sigma' \in \Sigma_G$

for which $\sigma \dot{\subset} \sigma'$ holds. If there is no such path, there are no $val_2$ and $order_2$ of $W$ that change the outcome of $\varphi$, and as a consequence AC2(2) is fulfilled by $\sigma$. If we do find such a path $\sigma'$, it contains all variables in $Z$ with $val_1$ and $order_1$ and some events $W$ with $val_2$ and $order_2$ that change the outcome of $\varphi$. In other words, the non-occurrence of the events in $W$ on $\sigma$ was causal for $\varphi$. In order to identify those events, we search for the minimal paths $R = \{\sigma' \in \Sigma_G | \sigma \dot{\subset} \sigma'\}$. For each path in $R$ we negate the events that are in $\sigma$ but not in $\sigma'$ and add them to the candidate set. Subsequently, we remove $\sigma$ from the candidate set.

Consider the case $Z = G \wedge O \wedge T \wedge C$ in our rail road crossing model. It is necessary that no event $G$ occurs between $O$ and $T$ for this ordered conjunction of events to lead to a hazard. If the system execution $G \wedge O \wedge G \wedge T \wedge C$ is possible, which means that there is a path representing this execution in the set *NCX(A)* for top level event $A$, we hence have to replace $Z$ by $Z' = G \wedge O \wedge \neg G \wedge T \wedge C$.

*Test for AC3:* Due to the minimality property of the candidate set, no test for AC3 is necessary.

*Test for OC1:* We need to decide whether for all ordered conjunctions in *CFT(T)* the order of the events is relevant to cause $T$. For each path $\sigma \in$ *CFT(T)*, we check whether the order of the events to occur is important or not. If the order of events in $\sigma$ is not important, then there has to be at least one path $\sigma' \in$ *CFT(T)* for which $\sigma = \sigma'$ and not $\sigma \doteq \sigma'$. For each event $e_i$ in $\sigma$ we check for all other events $e_j$ with $i < j$ whether $\sigma' \vDash e_i \wedge e_j$ for all $\sigma' \in$ *CFT(T)*. If $\sigma' \vDash e_i \wedge e_j$ for all $\sigma' \in$ *CFT(T)*, we mark this pair of events as having an order which is important for causality. If we can not find such a $\sigma'$, we mark the whole path $\sigma$ as having an order which is important for causality.

### 3.4 Adding Probabilities

In order to properly compute the probability mass that is to be attributed to the TLE $T$ in the fault tree it is necessary to account for all paths that can cause $T$. If there are two paths $\sigma_1, \sigma_2 \in \Sigma_C$ which, when combined, deliver a path $\sigma_{12} \in \Sigma_C$, then the probability mass of all three paths needs to be taken into account when calculating the probability for reaching $T$.

To illustrate this point, consider an extension of the railroad example introduced above. We add a traffic light indicating to the car driver that a train is approaching. Event $R$ indicates that the traffic light on the crossing is red, while the red light being off is denoted by event $L$. The top level event $A$ denoting the hazard is expressed as a state proposition applicable to the underlying stochastic model that states that neither the red light is on not the gate is closed, and that the train approaches and the car is in the crossing. Assume that the above described test would identify the following ordered conjunctions of events to be causal: $\neg G \wedge T \wedge C$ and $\neg L \wedge T \wedge C$. Due to the minimality property of *CFT(A)* the ordered conjunction $\neg G \wedge \neg L \wedge T \wedge C$ would be missing. We would hence lose the probability mass associated with the corresponding trace in the counterexample, as well as the qualitative information that the simultaneous failure of the red

light and the gate also leads to a hazardous state. To account for this situation we introduce the combination condition CC1.

CC1: Let $\sigma_1, \sigma_2, ...\sigma_k \in$ *CFT(L)* paths and $\psi_1, \psi_2, ..., \psi_k$ the event conjunctions representing them. A path $\sigma$ is added to *CFT(L)* if for $k \geq 2$ paths in *CFT(L)* it holds that $\sigma \vDash \psi_1 \wedge \sigma \vDash \psi_2 \wedge ... \wedge \sigma \vDash \psi_k$.

We can now assign each path in the candidate set the sum of the probability masses of the paths that it represents. This is done as follows: The probability of a path $\sigma_1$ in *CFT(L)* is the probability sum of all paths $\sigma'$ for which $\sigma_1$ is the only subset in in *CFT(L)*. The last condition is necessary in order to correctly assign the probabilities to paths which where added to the fault tree by test CC1.

All paths still in the candidate set are part of the fault tree and have now to be included in the fault tree representation. The fault trees generated by our approach all have a normal form, that is they start with an *intermediate*-gate representing the top level event, that is connected to an *OR*-gate. The paths in the candidate set *CFT(L)* will then be added as child nodes to the *OR*-gate as follows: Paths with a length of one and hence consisting of only one basic event are represented by the respective basic event. A path with length greater than one that has no subset of labels marked as ordered is represented by an *AND*-gate. This *AND*-gate connects the basic events belonging to that path. If a (subset of a) path is marked as ordered it is represented by a *PAND*-gate that connects the basic events in addition with an *Order Condition* connected to the *PAND*-gate constraining the order of the elements.

The probability values of the *AND*-gates are the corresponding probabilities of the paths that they represent. In order to display the probabilities in the graphical representation of the fault tree, we add an intermediate event as parent node for each *AND*-gate. The resulting intermediate events are then connected by an *OR*-gate that leads to the top event, representing the hazard. Since the path probabilities are calculated for a path starting from an initial state to the hazard state, the probability of the *OR*-gate is the sum of the probability of all child elements.

### 3.5 Algorithms and Complexity

Figure 4 depicts the fault tree generation algorithm. The methods called within the algorithm are depicted in Figures 5 to 14. The algorithm in Figure 4 shows how the tests defined in 3.3 are checked in the implementation. In order to give a worst case bound for the running time of our algorithm we determine the worst case bounds for all functions that are called within the algorithm. The runtime

of our algorithm can be expressed by

$$\begin{aligned}\mathrm{RT(CX2FT)} &= \mathrm{size(CX)} * \mathrm{RT(isMinimalPath)} * \mathrm{c} \\ &\quad + \mathrm{RT(checkAC22)} \\ &\quad + \mathrm{size(pathsCFT)} * \mathrm{RT(checkOC1)} \\ &\quad + \mathrm{size(pathsNotInFT)} * \mathrm{RT(checkCC1)} * \mathrm{RT(checkOC1)} * \mathrm{c'} \\ &\quad + \mathrm{RT(assignProbabilities)}\end{aligned}$$

where *RT(X)* denotes the run time of function X. The worst case execution time of our algorithm is

$$\begin{aligned}\mathrm{RT(CX2FT)} &= \mathrm{size(CX)}^2 + \mathrm{size(CX)} * \mathrm{size(G)} \\ &\quad + \mathrm{size(CX)}^2 + \mathrm{size(CX)}^3 + \mathrm{size(CX)}^2 \\ &\quad \in \mathrm{O}(\mathrm{size(CX)}^3 + \mathrm{size(CX)} * \mathrm{size(G)}).\end{aligned}$$

Note that we use size(X) to denote the number of elements in the list X. In the following size(events_p) denotes the number of events of the path $p$ in the counterexample ($CX$) that consists of the largest number of events. Since paths all paths are finite we can treat size(events_p) as a constant.

The function *isMinimalPath* depicted in Fig. 5 is used to compute the paths included in the candidate set (c.f. Definition 4). The worst case running time of the function*isMinimalPath* is

$$\begin{aligned}\mathrm{RT(isMinimalPath)} &= \mathrm{size(CX)} * \mathrm{size(events\_p)} * c \\ &\quad \in \mathrm{O}(\mathrm{size(CX)} * \mathrm{size(events\_p)}) \\ &\quad \in \mathrm{O}(\mathrm{size(CX)})\end{aligned}$$

The function *checkAC22* depicted in Fig. 6 is the implementation of the test of *AC2(2)*. The the running time of the function *checkAC22* is

$$\begin{aligned}\mathrm{RT(checkAC22)} &= \mathrm{size(pathsCFT)} * \mathrm{c} * \mathrm{size(G)} * \mathrm{size(events\_p)}^2 \\ &\quad + \mathrm{size(events\_p)}^2 + \mathrm{size(events\_p)} * \mathrm{c'} \\ &\quad \in \mathrm{O}(\mathrm{size(pathsCFT)} * \mathrm{size(G)} * \mathrm{size(events\_p)}^2) \\ &\quad \in \mathrm{O}(\mathrm{size(CX)} * \mathrm{size(G)}).\end{aligned}$$

Note that in the worst case *size(pathsCFT) = size(CX)* and *size(events_p)* denotes the number of events of the path $p$ in $CX$ with the most events. Since all paths are finite we can treat $size(events\_p)$ as a constant. We hence get *RT(checkAC22)* $\in$ *O(size(CX) * size(G))*.

The test of *OC1* is done by the function *checkOC1* depicted in Fig. 7. The running time of the function *checkOC1* is

$$\begin{aligned}\mathrm{RT(checkOC1)} &= \mathrm{size(events\_p)}^2 \\ &\quad + \mathrm{size(pathsCFT)} * \mathrm{c} * \mathrm{size(events\_p)}^2 * \mathrm{size(events\_p)} * \mathrm{size(events\_p)!} * \mathrm{c'} \\ &\quad \in \mathrm{O}(\mathrm{size(pathsCFT)} * \mathrm{size(events\_p)}^3 * \mathrm{size(events\_p)!}) \\ &\quad \in \mathrm{O}(\mathrm{size(CX)})\end{aligned}$$

```
INPUT: CX paths in counterexample,
G paths not in counterexample
(sorted by path length).

List pathsCFT;
List pathsNotInFT;

FOR ALL p in CX
{
        if(isMinimalPath(p))
        {
                pathsCFT.add(p);
        }
        else
        {
                pathsNotInFT.add(p);
        }
}

checkAC22();

FOR ALL p in pathsCFT
{
        checkOC1(p);
}

FOR ALL p in pathsNotInFT
{
        if(checkCC1(p))
        {
                checkOC1(p);
                p.setIsCC1(true);
                pathsCFT.add(p);
        }
}

assignProbabilities();
```

**Fig. 4.** Algorithm sketch

Note that in the worst case size(pathsCFT) = size(CX). Hence, we get RT(checkCC1) $\in$ *O(size(CX))*.

```
Input: path p
Output: true if p is minimal path

function boolean isMinimalPath(Path p)
{
        FOR ALL p' in CX
        {
                if(isRealSubset(p', p) && p = p')
                {
                        return false;
                }
        }
        return true;
}
```

**Fig. 5.** Algorithm sketch: isMinimalPath(p)

The function *checkCC1* depicted in Fig. 8 checks *CC1*. The the running time of the function *checkCC1* is

$$\begin{aligned} \mathrm{RT}(\mathrm{checkCC1}) &= \mathrm{size}(\mathrm{pathsCFT}) * \mathrm{size}(\mathrm{events\_p'})^2 * \mathrm{c} \\ &\quad + \mathrm{c'} + \mathrm{size}(\mathrm{pathsNotInFT}) * \mathrm{size}(\mathrm{events\_p''})^2 \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{CX}) * \mathrm{size}(\mathrm{events\_p'})^2). \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{CX})) \end{aligned}$$

Note that size(pathsCFT) + size(pathsNotInFT) = size(CX). Hence, we get *RT(checkCC1)* $\in$ *O(size(CX))*.

The function responsible for assigning the probabilities to the paths in the fault tree, *assignProbabilities*, is depicted in Fig. 9. The the running time of the function *assignProbabilities* is

$$\begin{aligned} \mathrm{RT}(\mathrm{assignProbabilities}) &= \mathrm{size}(\mathrm{pathsCFT}) * \mathrm{size}(\mathrm{CX}) * \mathrm{size}(\mathrm{events\_p})^2 * \mathrm{c} \\ &\quad + \mathrm{size}(\mathrm{pathsCFT}) \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{CX})^2). \end{aligned}$$

since each path in pathsCFT returns either *true* or *false* when *p.isCC1()* is called, the inner for loop is executed at most size(pathsCFT) times. The + size(pathsCFT) accounts for the cases where *p.isCC1()* returns *false*. Note that in the worst case scenario size(pathsCFT) = size(CX). Hence, we get *RT(assignProbabilities)* $\in$ *O(size(CX)*$^2$*)*.

The $\dot{\subset}$ operator is implemented by *isRealSubset* depicted in Fig. 10. The running time of the function *isRealSubset* is

$$\begin{aligned} \mathrm{RT}(\mathrm{isRealSubset}) &= \mathrm{c} + \mathrm{size}(\mathrm{events\_p}) * \mathrm{size}(\mathrm{events\_p}) \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{events\_p})^2) \end{aligned}$$

```
function checkAC22()
{
        FOR i = 0 to i < size(pathsCFT)
        {
                j = 0;

                p' = pathsCFT.get(i);
                p = G.get(j);
                while(j < size(G) && (length(p') < p.length()))
                {
                        j++;
                        p = G.get(j);
                }

                while(j < size(G) && !isSubset(p',p))
                {
                        j++;
                        p = G.get(j);
                }

                if(j < size(G) && isSubset(p', p))
                {

                        List events_p' = p'.getEvents();
                        List events_p = p.getEvents();

                        u = 0;
                        FOR x = 0 to length(p)
                        {
                                if(events_p'.get(u) = events_p.get(x)))
                                {//events are same move to next event
                                        u++;
                                }
                                else
                                {//events are different negate event x
                                        p'.negateEvent(x);
                                }
                        }
                        //Replace p' with p containing negated events
                        pathsCFT.set(i, p);
                }
        }
}
```

**Fig. 6.** Algorithm sketch: checkAC22()

since the for-loop is at most executed size(events_p) times. And in each iteration the function *eventExistsInPath* with run time *O(size(events_p))* is called.

```
function checkOC1(Path p)
{
        //mark all pairs as ordered
        FOR i = 0 to size(events_p)
        {
                FOR i = j to size(events_p)
                {
                        p.MarkOrdered(i,j,true);
                }
        }

        FOR ALL p' in pathsCFT
        {
                if (p != p')
                {
                        if(isEqual(p, p')
                        {
                                events_p = p.getEvents();
                                events_p' = p'.getEvents();

                                if(isOrderedEqual(dp, dpCompare))
                                {
                                        pathsCFT.remove(p);
                                }
                                else
                                {
                                  FOR i = 0 to size(events_p)
                                  {
                                    FOR i = j to size(events_p)
                                    {
                                       if(!(events_p'.indexOf(events_p.get(i))
                                          <= events_p'.indexOf(events_p.get(j))))
                                        {//pair i,j is not ordered
                                           p.MarkOrdered(i,j,false);
                                        }
                                    }
                                  }
                                }
                        }
                }
        }
}
```

**Fig. 7.** Algorithm sketch: checkOC1(Path p)

The $\dot{\subseteq}$ operator is implemented by *isSubset* depicted in Fig. 11. The the running time of the function *isSubset* is

$$\begin{aligned} \mathrm{RT}(\mathrm{isSubset}) &= \mathrm{c} + \mathrm{size}(\mathrm{events\_p}) * \mathrm{size}(\mathrm{events\_p}) \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{events\_p})^2) \end{aligned}$$

```
function boolean checkCC1(Path p)
{
        i = 0;
        FOR ALL p' in pathsCFT
        {
                if(isRealSubset(p', p))
                {
                        i++;
                }
        }

        if(i >= 2)
        {
                FOR ALL p'' in pathsNotInFT
                {
                        if(isSubset(p'', p) && p != p'')
                        {
                                return false;
                        }
                }
                return true;
        }
        return false;
}
```

**Fig. 8.** Algorithm sketch: checkCC1(Path p)

since the for-loop is at most executed *size(events_p)* times. And in each iteration the function *eventExistsInPath* with run time *O(size(events_p))* is called.

The function *eventExistsInPath* depicted in Fig. 12 checks whether the specified event occurs on the path. The running time of the function *eventExistsInPath* is

$$\begin{aligned} \mathrm{RT}(\mathrm{eventExistsInPath}) &= \mathrm{c} + \mathrm{size}(\mathrm{events\_p}) * \mathrm{c'} \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{events\_p})) \end{aligned}$$

since the for-loop is at most executed *size(events_p)* times.

The = operator is implemented by *isEqual* depicted in Fig. 13. The the running time of the function *isEqual* is

$$\begin{aligned} \mathrm{RT}(\mathrm{isEqual}) &= \mathrm{c} + \mathrm{size}(\mathrm{events\_p}) * \mathrm{size}(\mathrm{events\_p}) \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{events\_p})^2) \end{aligned}$$

since the for-loop is at most executed size(events_p) times. And in each iteration the function *eventExistsInPath* with run time *O(size(events_p))* is called.

```
Input: path p
Output: true if p is minimal path

function assignProbabilities();
{
        FOR ALL p in CFT
        {
                if(p.isCC1())
                {
                        FOR ALL p' in CX
                        {
                                if(isSubset(p,p'))
                                {
                                        p.Prob =  p.Prob + p'.Prob;
                                        p'.Prob = 0;
                                }
                        }
                }
        }
        FOR ALL p in CFT
        {
                if(!p.isCC1())
                {
                        FOR ALL p' in CX
                        {
                                if(isSubset(p,p'))
                                {
                                        p.Prob =  p.Prob + p'.Prob;
                                        p'.Prob = 0;
                                }
                        }
                }
        }
}
```

**Fig. 9.** Algorithm sketch: assignProbabilities()

The $\doteq$ operator is implemented by *isOrderedEqual* depicted in Fig. 14. The running time of the function *isOrderedEqual* is

$$\begin{aligned} \mathrm{RT}(\mathrm{isOrderedEqual}) &= \mathrm{c} + \mathrm{size}(\mathrm{events\_p}) * \mathrm{c'} \\ &\in \mathrm{O}(\mathrm{size}(\mathrm{events\_p})) \end{aligned}$$

since the for-loop is at most executed size(events_p) times.

```
function boolean isRealSubset(Path p', Path p)
{
        if(length(p') >= length(p))
        {
                return false;
        }

        List events = p1.getEvents();

        FOR ALL e in events
        {
                if(!eventExistsInPath(p, e))
                {
                        return false;
                }
        }

        return true;
}
```

**Fig. 10.** Algorithm sketch: isRealSubset(Path p', Path p)

```
function boolean isSubset(Path p', Path p)
{
        if(length(p') > length(p))
        {
                return false;
        }

        List events = p1.getEvents();

        FOR ALL e in events
        {
                if(!eventExistsInPath(p, e))
                {
                        return false;
                }
        }

        return true;
}
```

**Fig. 11.** Algorithm sketch: isSubset(Path p', Path p)

```
function boolean eventExistsInPath(Path p, event e)
{
        List events = p.getEvents();

        FOR ALL e' in events
        {
                if(e = e')
                {
                        return true;
                }
        }

        return false;
}
```

**Fig. 12.** Algorithm sketch: eventExistsInPath(Path p, event e)

```
function boolean isEqual(Path p', Path p)
{
        if(length(p') != length(p))
        {
                return false;
        }
        else
        {
                List events = p1.getEvents();

                FOR ALL e in events
                {
                        if(!eventExistsInPath(p, e))
                        {
                                return false;
                        }
                }
        }
        return true;
}
```

**Fig. 13.** Algorithm sketch: isEqual(Path p', Path p)

```
function boolean isOrderedEqual(Path p, Path p')
{
        if(length(p) != length(p'))
        {
                return false;
        }

        events_p = p.getEvents();
        events_p' = p'.getEvents();

        FOR i = 0 to size(events_p)
        {
                if(events_p.get(i) != events_p'.get(i)))
                {
                        return false;
                }
        }
        return true;
}
```

**Fig. 14.** Algorithm sketch: isOrderedEqual(Path p, Path p')

### 3.6 Scalability of the approach

Even though the worst case running time of the algorithm is in $O(size(CX)^3 + size(CX) * size(G))$ our case studies presented in Section 4 show that the fault tree computation finishes in several seconds, while the computation of the counterexample took several minutes. Hence, the limiting factor of our approach is the time needed for the computation of the counterexample.

## 4 Case Studies

In the following we present three case studies. The first two are taken from the literature, c.f. Section 4.1 and Section 4.2, while the second is an industrial case study, c.f. Section 4.3. Notice that we assume the PRISM models in practical usage scenarios to be automatically synthesized from higher-level design models, such as for instance by our QuantUM tool [25]. However, the case studies presented in this paper were directly modeled in the PRISM [22] language. We computed the counterexamples using our counterexample generation tool DiPro [3], which in turn uses the PRISM model checker.

### 4.1 Embedded Control System

This case study models an embedded control system based on the one presented in [29]. The system consists of a main processor (**M**), an input processor (**I**), an output processor (**O**), 3 sensors and two actuators. The input processor I reads data from the sensors and forwards it to M. Based on this data, M sends instructions to the output processor O which controls both actuators according to the received instructions.

Various failure modes, such as the failure of I, M or O, or sensor or actuator failures, can lead to a shutdown of the system. We are interested in generating the fault tree for the top level event "System shut down within one hour". One hour corresponds to 3,600 time units as we take one second as the basic time unit in our model. In CSL, this property reads as $\mathcal{P}_{=?}(true\ U^{\leq 3,600}\ down)$. We applied the XBF algorithm in order to generate counter examples for the property $\mathcal{P}_{=?}\ (true\ U^{\leq 3,600}\ down)$. The resulting counter example consists of 2024 paths. Figure 15 shows the fault tree generated by CX2FTA. The fault tree consists of 6 paths. The fault tree illustrates that the top level event down can be caused by a failure in the main processor (MainProcFail), a failure in the input/output processor (Input/OutputProcFail), a transient failure in the input processor (InputPtocTransFail) or the failing of sensors / actuators (SensorFailure and SensorFailure (1) / ActuatorFailure and ActuatorFailure (1)).

Figure 16 shows the memory and run time consumption of the fault counterexample and fault tree computation[4]. The computation of the fault tree is finished in several seconds, whereas the computation of the counterexample takes

[4] Experiments experiments where performed on a PC with an Intel Core 2 Duo processor with 3.06 Ghz and 8 GBs of RAM.

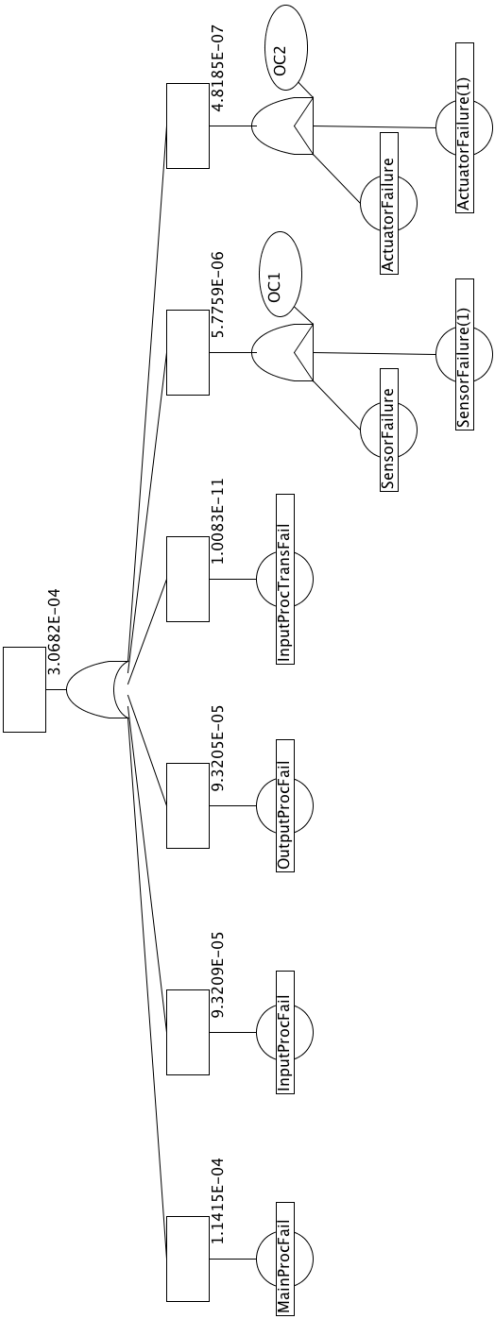


**Fig. 15.** Fault Tree of the Embedded Control System

| T (h) | Runtime CX (sec.) | Paths in CX | Mem. CX | Runtime FT | Paths in FT | Mem. FT |
|---|---|---|---|---|---|---|
| 1 | 1 703 (≈ 28.38 min.) | 2024 | 15.8 MB | 1.4 (sec.) | 6 | 25 MB |
| 10 | 2 327 (≈ 39 min.) | 2024 | 15.9 MB | 1.3 (sec.) | 6 | 25 MB |
| 100 | 3 399 (≈ 56.60 min.) | 2024 | 15.9 MB | 1.3 (sec.) | 6 | 26 MB |

**Fig. 16.** Experiment results of the Embedded Control System Case Study for T=1h, T=10h and T=100h.

several minutes. While the different running times of the counterexample computation algorithm seems to be caused by the different values of the mission time $T$, the variation of the running time of the fault tree computation seems to be caused by background processes on the experiment PC. The difference between memory used by the counterexample computation and the fault tree generation is caused by the fact that the counterexample generation tool stores the paths in a graph whereas the fault tree computation tool stores them individually.

### 4.2 Train Odometer Controller

This case study of a train odometer system taken from [9]. The train odometer system consists of two independent sensors used to measure the speed and position of a train. A wheel sensor is mounted to an unpowered wheel of the train to count the number of revolutions. A radar sensor determines the current speed by evaluating the Doppler shift of the reflected radar signal. We consider transient faults for both sensors. For example water on or beside the track could interfere with the detection of the reflected signal and thus cause a transient fault in the measurement of the radar sensor. Similarly, skidding of the wheel affects the wheel sensor. Due to the sensor redundancy the system is robust against faults of a single sensor. However, it needs to be detectable by other components in the train, when one of the sensors provides invalid data. For this purpose a monitor continuously checks the status of both sensors. Whenever either the wheel sensor or the radar sensor are failed, this is detected by the monitor and the corresponding status variable (wsensor or rsensor) is set to false. This information can be used by other train components that have to disregard temporary erroneous sensor data. Due to the robustness against single faults and since both sensor faults are transient the system even can recover completely from such a situation. If both sensors fail the monitor initiates an emergency brake maneuver, and the system is brought into a safe state. Only if the monitor fails, any subsequent faults in the sensors will no longer be detected. Since now the train may be guided by invalid speed and position information such situations are safety critical.

We generated the counterexample for the CSL formula

$$P_{=?}[(\text{true})U^{<=T}(\mathit{unsafe})]$$

where *unsafe* represents the above described unsafe state of the system and $T$ represents the mission time. We computed the probability for the mission

time T=10, T=100, and T=1000 and recorded the runtime for the counterexample computation (Runtime CX), the number of paths in the counterexample (Paths in CX), the runtime of the fault tree generation algorithm (Runtime FT) and the numbers of paths in the fault tree (Paths in FT) in Figure 18. Figure 17 shows the fault tree generated from the counterexample for the formula $P_{=?}[(\text{true})U^{<=10}(\mathit{unsafe})]$. While the counterexample consists of 108 paths, the fault tree comprises only 5 paths. In the fault tree it is easy to see that all paths contain the basic event *WAIT_MON_FAIL* and a number of basic events representing a failure of the wheel sensor, or of the radar sensor, or of both sensors. Again, if our fault tree method would not be used, the same conclusion would require to compare all 108 paths manually.

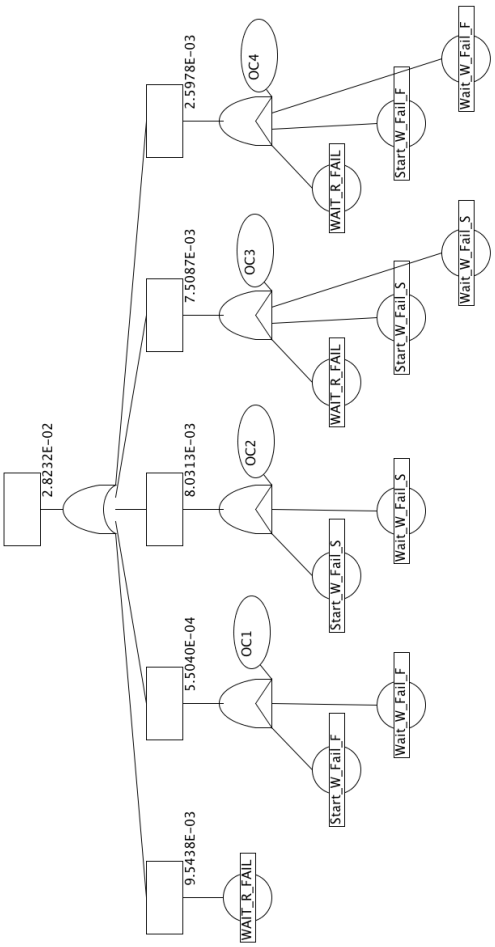


**Fig. 17.** Fault tree of the Train Odometer for T = 10

| T | Runtime CX (sec.) | Paths in CX | Mem. CX | Runtime FT | Paths in FT | Mem. FT |
|---|---|---|---|---|---|---|
| 10 | 433 (≈ 7.21 min.) | 108 | 78.00 MB | 7 .9 (sec.) | 5 | 122 MB |
| 100 | 582 (≈ 9.70 min.) | 108 | 77.90 MB | 8.0 (sec.) | 5 | 116 MB |
| 1000 | 1 298 (≈ 21.63 min.) | 108 | 78.20 MB | 8.1 (sec.) | 5 | 134 MB |

**Fig. 18.** Experiment results of the Train Odometer Case Study for T=10, T=100 and T=1000.

Figure 18 shows that the computation of the fault tree is finished in under one second, whereas the computation of the counterexample takes several minutes. While the different running times of the counterexample computation algorithm seems to be caused by the different values of the running time $T$, the variation of the running time of the fault tree computation seems to be caused by background processes on the experiment pc. The difference between memory used by the counterexample computation and the fault tree generation is caused by the fact that the counterexample generation tool stores the paths in a graph whereas the fault tree computation tool stores them individually. The compared with the other case studies high amount of memory needed for counterexample and fault tree generation, as well as the high run time of the fault tree computation algorithm is caused by the high number of paths (10 347) in $\Sigma_G$.

### 4.3 Airbag System

This case study is taken from [2] and models an airbag system. The architecture of this system was provided by our industrial partner TRW Automotive GmbH. Note that the used probability values are merely approximate "ballpark" numbers, since the real values are intellectual property of TRW Automotive GmbH and cannot be published. An airbag system can be divided into three major classes of components: sensors, crash evaluation and actuators. An impact is detected by acceleration sensors (front/rear/side impact) and additional pressure sensors (side impact). Angular rate or roll rate sensors are used to detect rollover accidents. The sensor information is evaluated by a microcontroller which decides whether the sensed acceleration corresponds to a crash situation or not. The deployment of the airbags is only activated if the microcontroller decides that there was indeed a critical crash.

The airbag system architecture that we consider consists of two acceleration sensors whose task it is to detect front or rear crashes, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag.

The deployment of the airbag is secured by two redundant protection mechanisms. The Field Effect Transistor (FET) controls the power supply for the airbag squibs. If the Field Effect Transistor is not armed, which means that the FET-Pin is not high, the airbag squib does not have enough electrical power to ignite the airbag. The second protection mechanism is the Firing Application Specific Integrated Circuit (FASIC) which controls the airbag squib. Only if it receives first an arm command from the FET and then a fire command

from the microcontroller it will ignite the airbag squib. Although airbags save lives in crash situations, they may cause fatal behavior if they are inadvertently deployed. This is because the driver may loose control of the car when this deployment occurs. It is therefore a pivotal safety requirement that an airbag is never deployed if there is no crash situation.

We are interested in generating the fault tree for an inadvertent ignition of the airbag, that is the ignition of the airbag although no crash occurred. In CSL, this property can be expressed using the formula $\mathcal{P}_{=?}(noCrash\ U^{\leq T}\ AirbagIgnited)$. We applied the XBF algorithm to this property in order to generate the respective counterexamples.

Figure 19 shows the fault tree generated by CX2FTA. For better readability we have omitted the order constraints of the *PAND*-gates. While the counterexample consists of 738 paths, the fault tree comprises only 5 paths. It is easy to see by which basic events, and with which probabilities, an inadvertent deployment of the airbag is caused. There is only one single fault that can lead to an inadvertent deployment, namely *FASICShortage*.

The basic event *MicroControllerFailure* can leads to a inadvertent deployment if it is followed by the following sequence of basic events: *enableFET*, *armFASIC*, and *fireFASIC*. This is an example where we over-approximate the set of causal events, *MicroControllerFailure* is the actual cause for the failure and the sequence *enableFET*, *armFASIC*, and *fireFASIC* represents the causal process. If the basic event *FETStuckHigh* occurs prior to the *MicroControllerFailure* the sequence *armFASIC*, and *fireFASIC* occurring after the *MicroControllerFailure* event suffices.

It is also easy to see that the combination of the basic events *FETStuckHigh* and *FASICStuckHigh* only lead to an inadvertent deployment of the airbag if the basic event *FETStuckHigh* occurs prior to the basic event *FASICStuckHigh*.

Figure 20 shows the memory and run time consumption of the fault counterexample and fault tree computation[5]. We recorded the runtime for the counterexample computation (Runtime CX), the number of paths in the counterexample (Paths in CX), the amount of memory consumed by the counterexample generation tool (Mem. CX), the runtime of the fault tree generation algorithm (Runtime FT), the numbers of paths in the fault tree (Paths in FT) and the memory consumed by the fault tree algorithm (Mem. FT). The computational effort is dominated by the counterexample computation. Increasing the parameter $t$ (mission time) in the process model has only a marginal influence on the computational effort needed.

[5] Experiments experiments where performed on a PC with an Intel Core 2 Duo processor with 3.06 Ghz and 8 GBs of RAM.

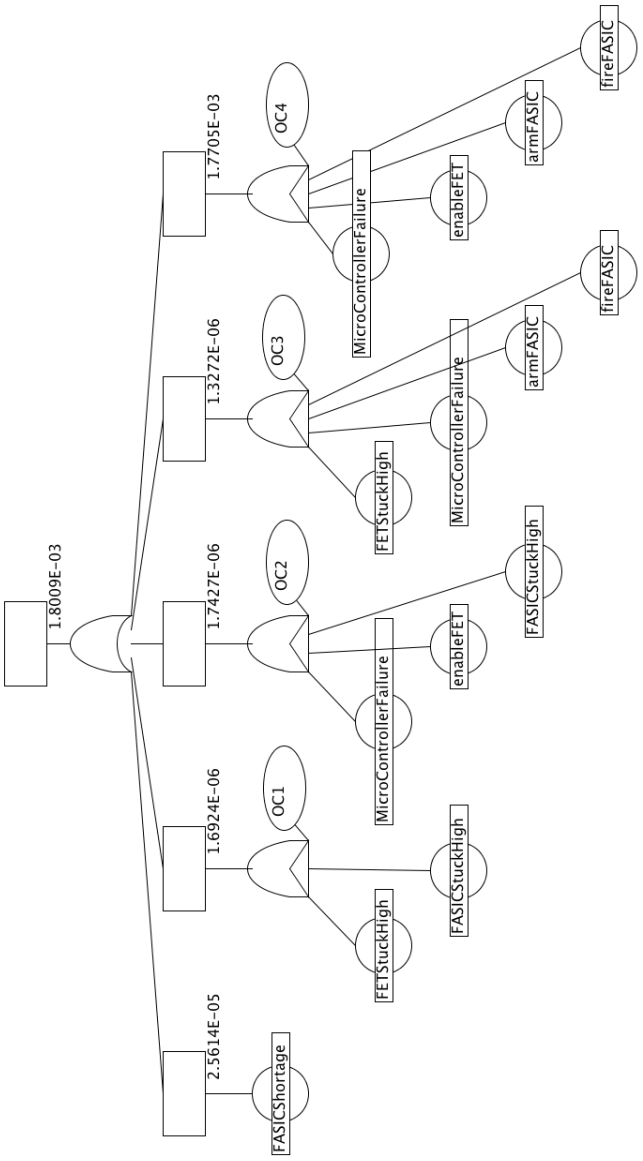


**Fig. 19.** Fault Tree of the Airbag System

| t | Runtime CX (sec.) | Paths in CX | Mem. CX | Runtime FT | Paths in FT | Mem. FT |
|---|---|---|---|---|---|---|
| 10 | 1 147 (≈ 19.12 min.) | 738 | 29.17 MB | 1.3 (sec.) | 5 | 27 MB |
| 100 | 1 148 (≈ 19.13 min.) | 738 | 29.20 MB | 1.3 (sec.) | 5 | 27 MB |
| 1000 | 1 263 (≈ 21.05 min.) | 738 | 29.49 MB | 1.8 (sec.) | 5 | 27 MB |

**Fig. 20.** Experiment results for T=10, T=100 and T=1000.

The case study shows that the fault tree is a compact and concise visualization of the counterexample which allows for an easy identification of the basic events that cause the inadvertent deployment of the airbag and their corresponding probabilities. If the order of the events is important, this can be seen in the fault tree by the *PAND*-gate, in the counterexample one would have to manually compare the order of the events in all 738 paths, which is a tedious and time consuming task.

## 5 Related Work

Work described in [10, 11] presents an interpretation of fault trees in terms of temporal logic and suggest algorithmic strategies for using fault trees as temporal specifications in model checking. In a similar sense, [32] proposes a formal semantics of fault trees based on an temporal logic interpretation. This is the opposite direction of what we aim to accomplish, namely to derive fault trees from system execution models.

Various approaches to derive fault trees semi-automatically or automatically from various semi-formal or formal models exist in the literature [30, 13, 31, 12, 27]. Contrary to our method, none of these methods uses sets of system execution sequences as the basis of the fault tree derivation, or provides an automatic probabilistic assessment of the synthesized fault tree nodes. These approaches also lack a justification of the causality model used.

Closest to our work is that described in [9]. The authors use minimal cut sets for the computation of the probabilities to reach a safety critical state. The system is modeled using extended Statecharts. Information about the failure transition labels, the failure transitions, and the relevant time bounds have to be provided either as an annotation to the Statecharts or as a separate specification in a language different than extended Statecharts. The fault configurations of the system that lead to a safety-critical state are represented as cut sets. Our work extends and improves on the approach of [9] in the following ways: (1) We use a single system specification and modeling language, namely the PRISM language, for describing the behavior of the system, the failure model and the relevant time bounds in an integrated fashion. The approach of [9] uses a large number of different tools and modeling notations. (2) By using the PRISM model checker and the counterexample computation capabilities that we integrated into PRISM, it is possible to use the full expressiveness of CSL for the specification of probabilistic measures of interest, whereas the approach of [9] only allows for the analysis of timed reachability properties. (3) Whereas in the approach of [9] only minimal cut sets are generated, we generate complete fault trees, thus providing more information to the user. (4) By allowing PAND-gates we support the full set of fault tree operators defined in [34], which is not the case for the approach of [9]. (5) Finally, we provide a justification of the causality model used in the fault tree derivation.

Work documented in [8] uses the Halpern and Pearl approach to determine causality for counterexamples in functional CTL model checking. However, this

approach considers only functional counterexamples that consist of single execution sequences.

We have given various references to work on the formal treatment of causality throughout the paper. It should be pointed out that [16] contains a careful analysis of the complexity of computing causality in the SEM. Most notable is the result that even for an SEM with only binary variables computing causal relationships between variables is NP-complete.

## 6 Conclusion

We presented a method and tool that automatically generates a fault tree from a probabilistic counterexample. Using three case studies we demonstrated that our approach improves and facilitates the analysis of safety critical systems. The resulting fault trees were significantly smaller and hence easier to understand than the corresponding stochastic counterexample, but still contain all information to discern the causes for the occurrence of a hazard.

The justification for the causalities determined by our method are based on an adoption of the Strucural Equation Model of Halpern and Pearl. We illustrated how to use this model in the analysis of computing systems and extended it to account for event orderings as causal factors. We presented an over-approximating implementation of the causality tests derived from this model. To the best of our knowledge this is the first attempt at using the structural equation model in this fashion.

In future work, we plan to further extend our approach, in particular to support the generation of dynamic fault-trees [15]. We are also interested in incorporating causality analysis directly into model checking algorithms.

**Acknowledgments:** The authors wish to thank Mark Burgess for giving them access to the FaultCAT source code.

## References

1. A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying Continuous-Time Markov Chains. In *Proc. of CAV 1996*, volume 1102 of *LNCS*, pages 269–276, New Brunswick, NJ, USA, 1996. Springer.
2. H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In *Proc. of QEST 2009*, pages 299–308. IEEE Computer Society, 2009.
3. H. Aljazzar and S. Leue. Debugging of Dependability Models Using Interactive Visualization of Counterexamples. In *Proc. of QEST 2008*, pages 189–198. IEEE Computer Society Press, 2008.
4. H. Aljazzar and S. Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Transactions on Software Engineering*, 2009.
5. H. Aljazzar and S. Leue. Generation of counterexamples for model checking of markov decision processes. *Quantitative Evaluation of Systems, International Conference on*, 0:197–206, 2009.

6. M. E. Andrés, P. R. D'Argenio, and P. van Rossum. Significant Diagnostic Counterexamples in Probabilistic Model Checking. In *Proc. of HVC 2008*, volume 5394 of *LNCS*, pages 129–148. Springer, 2008.
7. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(7), 2003.
8. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.
9. E. Böde, T. Peikenkamp, J. Rakow, and S. Wischmeyer. Model Based Importance Analysis for Minimal Cut Sets. In *Proc. of ATVA 2008*, volume 5311 of *LNCS*, pages 303 – 317. Springer, 2008.
10. M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *Proc. of ATVA 2007*, volume 4762 of *LNCS*, pages 162–176. Springer, 2007.
11. M. Bozzano and A. Villafiorita. Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform. In *Proc. of SAFECOMP 2003*, volume 2788 of *LNCS*, pages 49–62. Springer, 2003.
12. S. Cha, N. Leveson, and T. Shimeall. Safety verification in Murphy using fault tree analysis. In *Proc. of ICSE 1988*, pages 377–386. IEEE Computer Society Press, 1988.
13. B. Chen, G. Avrunin, L. Clarke, and L. Osterweil. Automatic Fault Tree Derivation From Little-Jil Process Definitions. In *Proc. of SPW/ProSim 2006*, volume 3966 of *LNCS*, pages 150–158. Springer, 2006.
14. J. Collins, editor. *Causation and Counterfactuals*. MIT Press, 2004.
15. J. Dugan, S. Bavuso, and M. Boyd. Dynamic Fault Tree Models for Fault Tolerant Computer Systems. *IEEE Transactions on Reliability*, 41(3):363–377, 1992.
16. T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. *Artificial Intelligence*, (1):53–89, 2002.
17. H. Fecher, M. Huth, N. Piterman, and D. Wagner. Hintikka games for PCTL on labeled Markov chains. In *Proc. of QEST 2008*, pages 169–178. IEEE Computer Society, 2008.
18. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Software Tools for Technology Transfer*, 8(3), 2006.
19. L. Grunske, R. Colvin, and K. Winter. Probabilistic model-checking support for fmea. In *Proc. of QEST 2007*, pages 119–128. IEEE Computer Society, 2007.
20. J. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 56(4):843, 2005.
21. T. Han, J.-P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. Softw. Eng.*, 35(2):241–257, 2009.
22. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *Proc. of TACAS 2006*, volume 3966 of *LNCS*, pages 441–444. Springer, 2006.
23. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The Ins and Outs of The Probabilistic Model Checker MRMC. In *Proc. of QEST 2009*, pages 167–176. IEEE Computer Society, 2009.
24. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

25. F. Leitner-Fischer and S. Leue. QuantUM: Quantitative safety analysis of UML models. In *In Proceedings of the Ninth Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, 2011.
26. D. Lewis. *Counterfactuals*. Wiley-Blackwell, 2001.
27. M. McKelvin Jr, G. Eirea, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. A Formal Approach to Fault Tree Synthesis for the Analysis of Distributed Fault Tolerant Systems. In *Proc. of EMSOFT 2005*, page 246. ACM, 2005.
28. P. Menzies. Counterfactual theories of causation. In *Stanford Encyclopedia of Philosophy*. Stanford University, 2008. http://plato.stanford.edu/entries/causation-counterfactual/.
29. J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability*, 1(2):9–20, July 1994.
30. G. Pai and J. Dugan. Automatic synthesis of dynamic fault trees from UML system models. In *Proc. of ISSRE 2002*, page 243. IEEE Computer Society, 2002.
31. V. Ratan, K. Partridge, J. Reese, and N. Levenson. Safety Analysis Tools for Requirements Specifications. Available from URL http://www.safeware-eng.com/system and software safety publications/SafAnTooReq.pdf.
32. G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proc. Integrated Design and Process Technology IDPT-2002*. Society for Design and Process Science, 2002.
33. M. Schmalz, D. Varacca, and H. Völzer. Counterexamples in Probabilistic LTL Model Checking for Markov Chains. In *Proc. of CONCUR 2009*, volume 5710 of *LNCS*, pages 587 – 602. Springer, 2009.
34. U.S. Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.
35. A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.