Technical Report soft-13-03, Chair for Software Engineering, University of Konstanz, Copyright by the Authors 2013

Probabilistic Fault Tree Synthesis using Causality Computation

Florian Leitner-Fischer and Stefan Leue

University of Konstanz, Germany

Abstract. In recent years, several approaches to generate probabilistic counterexamples have been proposed. The interpretation of probabilistic 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 extend the structural equation approach by Pearl and Halpern, which is based on Lewis counterfactuals, so that it can be applied to reason about causalities in a state-action trace model induced by a probabilistic counterexample. The causality relationships derived by the extended structural equation model are then mapped onto fault trees. We demonstrate the usefulness of our approach by applying it to a selection of case studies known from literature.

1 Introduction

In recent joint work [1] with our industrial partner TRW Automotive GmbH we have proven the applicability of probabilistic formal analysis techniques to safety analysis in an industrial setting. In [1] 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 probabilistic counterexample [2] helps to analyze the counterexamples, 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) [3] is a well-established industrial method and graphical notation 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. The main drawback of fault tree analysis is that it relies on the ability of the 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. Our interpretation of the causality expressed by a fault tree is based on the counterfactual notion of causality [4], which is a widely accepted interpretation of causality in the realm of technical systems. 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 [5].
- For this model we compute counterexamples for probabilistic properties of interest, representing system hazards, using the DiPro [6] counterexample generation tool that extends PRISM. 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 counterfactual model of causality due to Halpern and Pearl [7] that we modify and extend to be applicable to our setting.
- The path probabilities computed by the probabilistic 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[1].

All analysis steps are fully automated and do not require user intervention.

This paper extends and refines precursory work presented in [8] in the following ways:

- We present a theoretical extension of our precursory work where the previously informally described causality computation is now formally defined and correctness proofs are given.
- The definition of the syntax and semantics of the proposed event order logic are revised an extended.
- An additional test to handle causal non-occurrence of events is proposed.
- We demonstrate the applicability of the approach using two additional case studies.

Structure of the Paper In Section 2 we briefly introduce the concepts of counterexamples in probabilistic model checking and fault trees, and present a running example that we will use later in the paper to illustrate the introduced concepts and definitions. In Section 3 we describe the model of causality that we use, discuss how causality relationships can be formally established from counterexamples and show how these causality relationships can be mapped to fault

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

trees. In Section 4 we demonstrate our approach on case studies known from the literature. A discussion of related work follows in Section 5. Finally, Section 6 concludes the paper.

2 Preliminaries

2.1 Counterexamples in Probabilistic Model Checking

In probabilistic 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) [9]. Given an appropriate system model and a CSL property, probabilistic model checking tools such as PRISM [5] can verify automatically whether the model satisfies the property. Probabilistic model checkers do not automatically provide counterexamples, but the computation of counterexamples has recently been addressed in, amongst others, [10, 11].

Notion of Counterexamples 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 formulas of the form $\mathcal{P}_{\leq p}(\varphi)$, where φ is path formula specifying undesired behavior of the the system. A counterexample for an upper bounded property is a set Σ_C of paths leading from the initial state to a state satisfying φ such that the accumulated probability of Σ_C violates the probability constraint $\leq p$. If the CSL formula $\mathcal{P}_{=?}\ (\varphi)$ is used, the probability of the path formula φ to hold is computed and the counterexample contains all paths fulfilling φ. If a path σ fulfills the CSL formula φ we denote this by $\sigma \vDash_{CSL} \varphi$. The probability of the counterexample is computed using a probabilistic model checker, in our case PRISM. Notice that in the setting of this paper the counterexample is computed completely, i.e., all paths leading into the undesired system state are enumerated in the counterexample.

2.2 Fault Trees and Fault Tree Analysis

Fault trees (FTs) [3] 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 or specified by an order constraint connected to the *PAND*-gate.

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. 1. 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 [3].

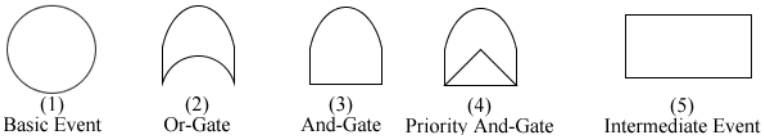

Fig. 1. Fault Tree Elements

2.3 Running Example

In the running example of a railroad crossing system that we will use in this paper, a train can approach the crossing (Ta), cross the crossing (Tc) and finally leave the crossing (Tl). Whenever a train is approaching, the gate should close (Gc) and will open when the train has left the crossing (Go). It might also be the case that the gate fails (Gf). The car approaches the crossing (Ca) and crosses the crossing (Cc) if the gate is open and finally leaves the crossing (Cl). We are interested in finding those events that lead to a hazard state in which both the car and the train are in the crossing at the same time. This hazard can be characterized by the CSL formula $\mathcal{P}_{=?}(\varphi)$ with $\varphi = \text{true}\,\mathcal{U}(\text{car\_crossing} \wedge \text{train\_crossing})$.

3 Computing Fault Trees from Counterexamples

In this section we discuss how fault trees can be computed from probabilistic counterexamples. In Section 3.1 we discuss how causality relationships can be inferred in general. The causality model that we present in Section 3.2 allows for causality computation on probabilistic counterexamples and takes the possible causality of event orders into account. How this causality model is used to generate fault trees from probabilistic counterexamples is shown in Section 3.3. Finally, we discuss the scalability and complexity of the proposed approach in Section 3.4.

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. Each path in the counterexample is a linearly ordered, interleaved sequence of concurrent events. 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 *actual cause* conditions as proposed by Halpern and Pearl [7] to determine the causality of sequences of events. Their definitions are based on *counterfactual* reasoning and the related *alternative world* semantics of Lewis [4, 12]. The counterfactual argument is widely used as the foundation for identifying faults in program debugging [13] and also underlies the formal fault tree semantics proposed in [14]. 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 logical conditions on the 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., [12]. Since we are considering concurrent systems in which particular event interleavings such as race conditions may be the cause of errors, the order of occurrence of events is a potential causal factor that cannot be disregarded. Consider the railroad crossing example introduced in Section 2.3. A naive counterfactual test will fail to show that the event sequence "Cc, Gc, Tc, Tl, Go" is a potential cause of a hazard, whereas "Gc, Tc, Tl, Go, Cc" is not. For the counterfactual argument to be applicable the constraint that whenever the causal events occur the hazard occurs as well has to hold. But on the above traces the same events occur, but the hazard occurs only on one trace.

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 [7] to what they refer to as *structural equation model* (SEM). It encompasses the notion of actual 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 [7] for the purpose of our paper. An actual cause is a cause from which irrelevant events have been removed. A causal formula is a boolean conjunction ψ 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[2].

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" ([7]) 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 ψ. 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_{V_1} that assigns values to all variables in V. In the alternate world, there is a function val_{V_2} assigning potentially different values to the variables in V. The causal process comprises all variables that mediate between X and the effect φ. Those variables are not root causes, but they contribute to rippling the causal effect through the system until reaching the final effect.

In the SEM, a formula ψ is an actual cause for an event represented by the formula φ, if the following conditions are met:

AC1: Both ψ and φ are true in the actual world, assuming the context defined by the variables in U, and given a valuation val_{V_1}.

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_{X_2} and val_{W_2} 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_{X_1} and val_{W_1} to val_{X_2} and val_{W_2} changes φ from true to false.
2. Setting the values of the variables in W from val_{W_1} to val_{W_2} should have no effect on φ as long as the values of the variables in X are kept at the values defined by val_{X_1}, even if all the variables in an arbitrary subset of $Z \setminus X$ are set to their value according to val_{Z_1}.

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

[2] Notice that the use of this operational form of event semantics makes the use of structural equations to define events as in [7] dispensable. In other words, we inherit from the SEM in [7] the general ideas of the actual cause definitions, but not the structural equation based event semantics.

AC2(1) corresponds to the Lewis counterfactual test. However, as [7] 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 φ in AC2(1) are considered part of the cause; inessential elements are pruned.

3.2 Inferring Causality from Probabilistic Counterexamples

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 formulas, referred to as structural equations. In these formulas, 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 probabilistic model checking. Technical systems evolve in discrete computation steps. A system state s is defining a valuation of the system state variables. 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 $s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ 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 σ_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, Act contains the events that we wish to reason about. We also assume that there exists a set $\mathcal{A}$ of event variables that contains a boolean variable for each action $\alpha \in Act$. For a given state-action trace σ, $L(s_i)$ contains the event variable a_{α_i} corresponding to the event α_i. The event variables are needed to formally express the occurrence of events in event order logic formulas. Notice that we consider event instances, not types.

In other words, the *i-th* occurrence of some action of type α will be represented by a distinct boolean variable from the *i+1st* occurrence of this event type. We denote the variable that is representing the event $\alpha_i \in Act$ by a_{α_i}.

Definition 2. *Event Types, Occurrence of Events, and Event Variables. Let* $M = (S, Act, AP, L, \Sigma)$ *a SATM and* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a state-action trace of M. We define the following:*

- *Each* $\alpha \in Act$ *defines an* event type α.
- α_i *of* σ *is the* i-th *occurrence of an event of the* event type α.
- *The variable representing the occurrence of event* α_i *is denoted by* a_{α_i}.
- *The set* $\mathcal{A} = \{a_{\alpha_1}, ..., a_{\alpha_n}\}$ *contains a boolean variable for each occurrence of an event.*

We next define an event order logic allowing us to reason about boolean conditions on the occurrence of events. The event order logic allows to connect event variables from $\mathcal{A}$ with the boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events we introduce the ordered conjunction operator $\owedge$. The formula $a \owedge b$ with $a, b \in \mathcal{A}$ is satisfied if and only if events a and b occur in a trace and a occurs before b. In addition to the $\owedge$ operator we introduce the interval operators $\owedge_[$, $\owedge_]$, and $\owedge_< \phi \owedge_>$, which define an interval in which an event has to hold in all states.

Definition 3. *Syntax of Event Order Logic. Simple event order logic formulas over the set* $\mathcal{A}$ *of event variables are formed according to the following grammar given in BNF like syntax rules:*

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 \vee \phi_2$$

where $a \in \mathcal{A}$ *and* ϕ, ϕ_1 *and* ϕ_2 *are simple event order logic formulas. Complex event order logic formulas are formed according to the following grammar given in BNF like syntax rules:*

$$\psi ::= \phi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \owedge \psi_2 \mid \psi \owedge_[\phi \mid \phi \owedge_] \psi \mid \psi_1 \owedge_< \phi \owedge_> \psi_2$$

where ϕ *is a simple event order logic formula.*

The formal semantics of this logic is defined on SATMs. Notice that the $\owedge$, $\owedge_[$, $\owedge_]$, and $\owedge_< \phi \owedge_>$ operators are temporal logic operators and that the SATMs model is akin to a linearly ordered Kripke structure.

Definition 4. *Semantics of Event Order Logic. Let* $M = (S, Act, AP, L, \Sigma)$ *a SATM, and* ϕ, ϕ_1, ϕ_2 *simple event order logic formulas and* ψ, ψ_1, ψ_2 *complex event order logic formulas, and let* $\mathcal{A}$ *a set of event variables, with* $a_{\alpha_i} \in \mathcal{A}$, *over which* ϕ, ϕ_1, ϕ_2 *are built. Let* $\sigma = s_0, a_1, s_1, a_2, \ldots a_n, s_n$ *a state-action trace over* (S, Act) *and let* $\sigma[i..r] = s_i, a_{i+1}, s_{i+1}, \ldots a_r, s_r$ *a partial state-action trace. We define that an event order logic formula is satisfied in a state* s_i *of* σ, *written* $s_i \vDash \psi$, *as follows:*

- $s_j \vDash a_{\alpha_j}$ iff $a_{\alpha_j} \in L(s_j)$
- $s_j \vDash \neg\phi$ iff not $s_j \vDash \phi$
- $\sigma[i..r] \vDash \phi$ iff $\exists j : i \leq j \leq r \;.\; s_j \vDash \phi$
- $\sigma \vDash \psi$ iff $\sigma[0..n] \vDash \psi$, *where* n *is the length of* σ.
- $\sigma[i..r] \vDash \phi_1 \wedge \phi_2$ iff $\sigma[i..r] \vDash \phi_1$ *and* $\sigma[i..r] \vDash \phi_2$
- $\sigma[i..r] \vDash \phi_1 \vee \phi_2$ iff $\sigma[i..r] \vDash \phi_1$ *or* $\sigma[i..r] \vDash \phi_2$
- $\sigma[i..r] \vDash \psi_1 \wedge \psi_2$ iff $\sigma[i..r] \vDash \psi_1$ *and* $\sigma[i..r] \vDash \psi_2$
- $\sigma[i..r] \vDash \psi_1 \vee \psi_2$ iff $\sigma[i..r] \vDash \psi_1$ *or* $\sigma[i..r] \vDash \psi_2$
- $\sigma[i..r] \vDash \psi_1 \wedge \psi_2$ iff $\exists j, k : i \leq j < k \leq r \;.\; \sigma[i..j] \vDash \psi_1$ *and* $\sigma[k..r] \vDash \psi_2$
- $\sigma[i..r] \vDash \psi \wedge_{[} \phi$ iff $(\exists j : i \leq j \leq r \;.\; \sigma[i..j] \vDash \psi$ *and* $(\forall k : j \leq k \leq r \;.\; \sigma[k..k] \vDash \phi))$
- $\sigma[i..r] \vDash \phi \wedge_{]} \psi$ iff $(\exists j : i \leq j \leq r \;.\; \sigma[j..r] \vDash \psi$ *and* $(\forall k : 0 \leq k \leq j \;.\; \sigma[k..k] \vDash \phi))$
- $\sigma[i..r] \vDash \psi_1 \wedge_{<} \phi \wedge_{>} \psi_2$ iff $(\exists j, k : i \leq j < k \leq r \;.\; \sigma[i..j] \vDash \psi_1$ *and* $\sigma[k..r] \vDash \psi_2$ *and* $(\forall l : j \leq l \leq k \;.\; \sigma[l..l] \vDash \phi))$

We define that the state-action trace model M *satisfies the formula* ψ*, written as* $M \vDash \psi$*, iff* $\exists \sigma \in M \;.\; \sigma \vDash \psi$.

Each state-action trace σ specifies an assignment of the boolean values *true* and *false* to the variables in the set $\mathcal{A}$. If an event α_i occurs on σ its value is set to *true*. If the event does not occur on σ its value is set to *false*, respectively. We define a function $val_{\mathcal{A}}(\sigma)$ that represents the valuation of all variables in $\mathcal{A}$ for a given σ.

Definition 5. *Valuation of a Set of Event Variables. Let* $M = (S, Act, AP, L, \Sigma)$ *a SATM,* σ *a state-action trace of* M *and* $\mathcal{A}$ *the set of event variables then we define the function* $val_{\mathcal{A}}(\sigma)$ *as follows:*

$$val_{\mathcal{A}}(\sigma) = (a_{\alpha_1}, ..., a_{\alpha_n}) \mid a_{\alpha_i} = \begin{cases} \text{true } \textit{if } \sigma \vDash a_{\alpha_i} \\ \text{false, } \textit{else} \end{cases} .$$

Further we define $val_{\mathcal{A}}(\sigma) = val_{\mathcal{A}}(\sigma')$ *if for all* $a_{\alpha_i} \in \mathcal{A}$ *the values assigned by* $val_{\mathcal{A}}(\sigma)$ *and* $val_{\mathcal{A}}(\sigma')$ *are equal and* $val_{\mathcal{A}}(\sigma) \neq val_{\mathcal{A}}(\sigma')$ *else.*

We partition the set of event variables $\mathcal{A}$ into sets Z and W. The events represented by the variables in Z are those events that are considered to be part of the causal process.

We can use an event order logic formula ψ over the variables in Z to define the order and occurrence of events on a causal process.

Definition 6. *Event Order Logic over State-Action Traces. Let* $M = (S, Act, AP, L, \Sigma)$ *a SATM,* σ *a state-action trace of* M*. The event order logic over the state-action trace* σ *denoted by* ψ_σ *is defined as follows: We partition the set of event variables* $\mathcal{A}$ *into sets* Z *and* W *in such a way that* Z *contains all event variables of the events that occur on* σ *and* W *contains all event variables of the events that do not occur on* σ*.* ψ_σ *is the event order logic formula containing all events in* Z *in the order they occur on* σ *(e.g.* $\psi_\sigma = a_{\alpha_1} \wedge a_{\alpha_2} \wedge ... \wedge a_{\alpha_n}$*).*

In Section 3.1 we already described that a causal process comprises all variables that mediate between the root causes and the effect or hazard.

We now present an adaption of the SEM that can be used to decide whether a given ψ describes the causal process of a hazard in a state-action trace model. If ψ describes the causal process of a hazard described by the CSL formula φ we also say ψ is causal for the hazard.

Definition 7. *Cause for a Hazard. Let $M = (S, Act, AP, L, \Sigma)$ a SATM, and σ, σ' and σ'' state-action traces of M. Let φ a CSL formula and $\sigma \vDash_{CSL} \varphi$ is true if φ is satisfied by σ. We partition the set of event variables $\mathcal{A}$ into sets Z and W. An event order logic formula ψ consisting of the event variables in Z is considered a cause for a hazard described by the CSL formula φ, if the following conditions are satisfied:*

AC1: *There exists a state-action trace σ, for which both $\sigma \vDash \psi$ and $\sigma \vDash_{CSL} \varphi$ hold.*

AC2 (1): *$\exists \sigma'$ s.t. $\sigma' \nvDash \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ and $\sigma' \nvDash_{CSL} \varphi$. In words, there exists a state-action trace σ' where the order and occurrence of events is different from the state-action trace σ and the hazard described by φ does not occur on σ'.*

AC2 (2): *$\forall \sigma''$ with $\sigma'' \vDash \psi \wedge (val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma''))$ it holds that $\sigma'' \vDash_{CSL} \varphi$ for all subsets of W. In words, for all state-action traces where the events in Z have the value defined by $val_Z(\sigma)$ and the order defined by ψ, the value and order of an arbitrary subset of the events in W have no effect on the occurrence φ.*

AC3: *The event order logic formula ψ is minimal: no subset of ψ satisfies conditions AC1 and AC2.*

If AC1 and AC2(1) are fulfilled but AC2(2) fails for a event order logic formula ψ_σ representing the events occurring on a state-action trace σ this means that at least one event α occurs on σ'' which did not occur on σ and the occurrence of α prevents the hazard. Hence, we need to reflect the causality of the non-occurrence of α in ψ_σ. For the models that we analyze there are three possibilities for such a preventing event α to occur, namely at the beginning of the state-action trace, at the end of the state-action trace, or between two other events α_1 and α_2. It is possible that the hazard is prevented by more than one event, hence we need to find the minimal set of events that are needed to prevent the hazard. This is achieved by finding the minimal sub-set $Q \subseteq W$ of event variables that need to be changed in order to prevent the hazard.

Definition 8. *Causal Non-Occurrence of Events. Let $M = (S, Act, AP, L, \Sigma)$ a SATM, and σ and σ'' state-action traces of M. We partition the set of event variables $\mathcal{A}$ into sets Z and W. Let ψ a event order logic formula consisting of the event variables in Z. The non-occurrence of the events which are represented by the event variables $a_\alpha \in Q$ with $Q \subseteq W$ on trace σ is causal for the hazard described by the CSL formula φ if the following is satisfied: ψ satisfies AC1 and AC2(1) but violates AC2(2) and Q is minimal, which means that there*

is no true subset of Q for which $\sigma'' \vDash \psi \wedge val_Z(\sigma) = val_Z(\sigma'') \wedge val_Q(\sigma) \neq val_Q(\sigma'') \wedge val_{W \setminus Q}(\sigma) = val_{W \setminus Q}(\sigma'')$ and $\sigma'' \vDash_{CSL} \varphi$ holds.

For each event variable $a_\alpha \in Q$ we determine the location of the event in ψ'' and prohibit the occurrence of α in the same location in ψ. We add $\neg a_\alpha \barwedge_]$ at the beginning of ψ if the event occurred at the beginning of σ'' and $\barwedge_[\neg a_\alpha$ at the end of ψ if the event occurred at the end of σ''. If α occurred between the two events α_1 and α_2 we insert $\barwedge_< \neg a_\alpha \barwedge_>$ between the two event variables a_{α_1} and a_{α_2} in ψ. Additionally, each event variable in Q is added to Z. After ψ was modified the conditions AC1, AC2(1), AC2(2) and AC3 are checked for the modified ψ.

If a formula ψ meets conditions AC1 through AC3, the occurrence of the events included in ψ is causal for the hazard described by φ. However, condition AC2 does not imply that the order of the occurring events is causal. If the order of the events is not causal, then there has to exist a state-action trace for each ordering of the events that is possible in the system, and the hazard occurs on all these state-action traces.

Definition 9. *Order Condition (OC1). Let $M = (S, Act, AP, L, \Sigma)$ a SATM, and σ, σ' state-action traces of M. Let ψ an event order logic formula over Z that holds for σ and let $\psi_\wedge$ the event order logic formula that is created by replacing all $\barwedge$-operators in ψ by $\wedge$-operators. The $\barwedge_[$, $\barwedge_]$, and $\barwedge_< \phi \barwedge_>$ are not replaced in $\psi_\wedge$.*

OC1: The order of a sub-set of events $Y \subseteq Z$ represented by the event order logic formula χ is not causal if the following holds: $\sigma \vDash \chi \wedge (\exists \sigma' \in \Sigma_B : \sigma' \nvDash \chi \wedge \sigma \vDash \chi_\wedge)$.

We will now demonstrate the above given definitions on the running example introduced in Section 2.3. In the following we will use short-hand notation $\sigma =$ "a_{α_1},a_{α_2},...,a_{α_n}" for a state-action trace $\sigma = s_0, \alpha_1, s_1, \alpha_2, ..., \alpha_n, s_n$. Suppose we want to check whether the events on the state-action trace $\sigma =$ "Ta, Ca, Gf, Cc, Tc" are causal for the hazard described by φ. We partition the set $\mathcal{A}$ of event variables in the set Z containing all the event variables of the events that occur on σ and the set W containing all the event variables of the events that do not occur on σ. The resulting event order logic formula over Z, that we want to show is causal, is $\psi_\sigma = \mathrm{Ta} \barwedge \mathrm{Ca} \barwedge \mathrm{Gf} \barwedge \mathrm{Cc} \barwedge \mathrm{Tc}$. Now we need to show that AC1, AC2(1), AC2(2) and AC3 are fulfilled for ψ_σ.

- AC1 is fulfilled, since there exists a state-action trace $\sigma =$ "Ta, Ca, Gf, Cc, Tc" for which $\sigma \vDash \psi_\sigma$, and both the train and the car are in the crossing at the same time.
- AC2(1) is fulfilled since there exists a state-action trace $\sigma' =$ "Ta, Ca, Gc, Tc" for which $\sigma' \nvDash \psi_\sigma \wedge (val_Z(\sigma) \neq val_Z(\sigma') \wedge val_W(\sigma) \neq val_W(\sigma'))$ holds and the hazard does not occur on σ'.
- Now we need to check the condition AC2(2). For the state-action trace $\sigma'' =$"Ta, Ca, Gf, Cc, Cl, Tc" and the partition $Z, W \subseteq \mathcal{A}$ $\sigma'' \vDash \psi_\sigma$ and $val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma'')$ holds. Since the car leaves the crossing (Cl) before the train enters the crossing (Tc) the hazard does

not occur. Hence AC2(2) is not fulfilled by ψ_σ. Cl is the only event that can prevent the property violation on σ and occurs between the events Cc and Tc. Consequently $\neg Cl$ is added to Z and ψ_σ, we hence get ψ_σ = Ta $\wedge$ Ca $\wedge$ Gf $\wedge$ Cc $\wedge_<$ $\neg$Cl $\wedge_>$ Tc.
- In our model there does not exist an event order logic formula that is a subset of ψ and satisfies conditions AC1 and AC2. As a consequence, AC3 is fulfilled.

Finally we need to check whether the order of events is causal with OC1.

In our example, the order of the events Gf, Cc, $\neg$Cl, Tc is causal since only if the gate fails before the car and the train are entering the crossing, and the car does not leave the crossing before the train is entering the crossing an accident happens. Consequently after OC1 we obtain the EOL formula ψ = Gf $\wedge$ ((Ta $\wedge$ (Ca $\wedge$ Cc)) $\wedge_<$ $\neg$Cl $\wedge_>$ Tc).

The following section demonstrates how the definitions from above can be used to generate a fault tree from a probabilistic counterexample.

3.3 Fault Tree Generation

In order to automatically synthesize a fault tree from a probabilistic counterexample, the combinations of basic events causing the top level event in the fault tree have to be identified. Using a probabilistic model checker we compute a counterexample which contains all paths leading to a state corresponding to the occurrence of some top level event φ. This is achieved by computing the counterexample for the CSL formula $\mathcal{P}_{=?}(\text{true}\ \mathcal{U}\ \varphi)$. We interpret counterexamples in the context of an SATM $M = (S, Act, AP, L, \Sigma)$. We assume that Σ, the set of all good and bad traces of the model, is partitioned in disjoint sets Σ_G and Σ_C. The set Σ_C contains all traces belonging to the counterexample, and the set Σ_G contains all system traces that do not belong to the counterexample.

Definition 10. *Good and Bad Traces. Let Σ the set of all good and bad traces of the model and φ the hazard. $\Sigma_G = \{\sigma \in \Sigma \mid \sigma \not\models_{CSL} \varphi\}$, $\Sigma_C = \{\sigma \in \Sigma \mid \sigma \models_{CSL} \varphi\}$, and $\Sigma_G \cup \Sigma_B = \Sigma$ and $\Sigma_G \cap \Sigma_B = \varnothing$.*

The disjointness of Σ_C and Σ_G implies that M is deterministic with respect to the causality of φ. 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. Without loss of generality we assume that for every trace $\sigma \in M_C$ the last state $s_n \models_{CSL} \varphi$, which implies that $M_c \models_{CSL} \varphi$. In our interpretation of the SEM, actual world models will be derived from Σ_C, whereas alternate world models are part of Σ_G. Notice that in order to compute the full model probability of reaching the stater where φ holds it is necessary to perform a complete state space exploration of the model that we analyze. We hence obtain M_G at no additional cost.

The key idea of the proposed algorithm is that the conditions AC1, AC2(1), AC2(2), and AC3 defined in Definition 7 can be mapped to computing sub-

and super-set relationships. In the following we also use the terms sub-trace and super-trace to refer to sub- or super-set relationships between different traces. In order to establish sub- and super-set relationships between traces we define a number of trace comparison operators.

Definition 11. *Trace Comparison Operators. Let* $M = (S, Act, AP, L, \Sigma)$ *a SATM, and* σ_1, σ_2 *state-action traces of M.*

$=$**:** $\sigma_1 = \sigma_2$ *iff* $\forall a \in \mathcal{A} \,.\, \sigma_1 \vDash a \equiv \sigma_2 \vDash a$.
$\doteq$**:** $\sigma_1 \doteq \sigma_2$ *iff* $\forall a_1, a_2 \in \mathcal{A} \,.\, \sigma_1 \vDash a_1 \wedge a_2 \equiv \sigma_2 \vDash a_1 \wedge a_2$.
$\subseteq$**:** $\sigma_1 \subseteq \sigma_2$ *iff* $\forall a \in \mathcal{A} \,.\, \sigma_1 \vDash a \Rightarrow \sigma_2 \vDash a$.
$\subset$**:** $\sigma_1 \subset \sigma_2$ *iff* $\sigma_1 \subseteq \sigma_2$ *and not* $\sigma_1 = \sigma_2$.
$\dot{\subseteq}$**:** $\sigma_1 \dot{\subseteq} \sigma_2$ *iff* $\forall a_1, a_2 \in \mathcal{A} \,.\, \sigma_1 \vDash a_1 \wedge a_2 \Rightarrow \sigma_2 \vDash a_1 \wedge a_2$.
$\dot{\subset}$**:** $\sigma_1 \dot{\subset} \sigma_2$ *iff* $\sigma_1 \dot{\subseteq} \sigma_2$ *and not* $\sigma_1 \doteq \sigma_2$.

We next define the candidate set of traces that we consider to be causal for φ. We define this set in such a way that it includes all minimal traces. Traces are minimal if they do not contain a sub-trace according to the $\subseteq$ operator that is also a member of the candidate set.

Definition 12 (Candidate Set). *Let* $M_C = (S, Act, AP, L, \Sigma_C)$ *a counterexample model, and* φ *a top level event in* M_C*. We define the candidate set of traces belonging to the fault tree of* φ *as CFT(*φ*):* $CFT(\varphi) = \{\sigma \in \Sigma_C | \forall \sigma' \in \Sigma_C \,.\, \sigma' \subseteq \sigma \Rightarrow \sigma' = \sigma\}$.

Notice that the candidate set is minimal in the sense that removing an event from some trace in the candidate set means that the resulting trace is no longer in the counterexample Σ_C. Given a counterexample model M_C, we state the following observations regarding the traces included in Σ_C: Each $\sigma \in \Sigma_C$ can be represented by an event order logic formula ψ_σ (c.f. Def. 6). And on each $\sigma \in \Sigma_C$, 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 trace and as a result the trace 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. Instead of computing the set X of events that are causal for some φ, we compute the set Z of events, which consists of all events that are part of the causal process of φ. Z will then be represented by ψ. 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 over-approximation 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 but not root causes, and hence mediate between X and φ. 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 show that AC1 is fulfilled by all ψ_σ with $\sigma \in CFT(\varphi)$.

Theorem 1. *AC1 holds for all ψ_σ with $\sigma \in CFT(\varphi)$.*

Proof. According to Definition 12 all traces $\sigma \in CFT(\varphi)$ are traces in Σ_C consequently $\sigma \vDash_{CSL} \varphi$ holds for all $\sigma \in CFT(\varphi)$. And $\sigma \vDash \psi_\sigma$ holds by definition of ψ_σ. Therefore AC1 holds for all ψ_σ with $\sigma \in CFT(\varphi)$.

In order to show that AC2(1) holds for all ψ_σ with $\sigma \in CFT(\varphi)$ we first demonstrate that AC2(1) is fulfilled for a σ if we can find a $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$.

Theorem 2. *AC2(1) holds for ψ_σ if there is a trace $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$.*

Proof. To show AC2(1) for a trace σ we need to show that there exists a trace σ' for which $\sigma' \nvDash \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ and $\sigma' \nvDash_{CSL} \varphi$ holds. For each $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$ there is at least one event on σ that does not occur on σ'. Because that missing event is part of ψ_σ and Z it follows $\sigma' \nvDash \psi_\sigma$ and $(val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ follows, since the value of the event variable representing the missing event assigned by $val_Z(\sigma)$ is *true* and the value assigned by $val_Z(\sigma')$ is *false*. Therefore, we can show AC2(1) for ψ_σ by finding a trace $\sigma' \in \Sigma_G$ for which $\sigma' \sqsubset \sigma$ holds.

We are now ready to show that AC2(1) holds for all ψ_σ with $\sigma \in CFT(\varphi)$.

Theorem 3. *AC2(1) holds for all ψ_σ with $\sigma \in CFT(\varphi)$.*

Proof. According to Theorem 2 AC2(1) holds for ψ_σ if there is a trace $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$. Recall that all traces in $CFT(\varphi)$ are minimal bad traces, that is all sub-traces of the traces in $CFT(\varphi)$ are good traces. Consequently, AC2(1) is fulfilled by all ψ_σ with $\sigma \in CFT(\varphi)$.

We now need to test AC2(2) for σ. AC2(2) requires that $\forall \sigma''$ with $\sigma'' \vDash \psi_\sigma \wedge (val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma''))$ holds $\sigma'' \nvDash_{CSL} \varphi$ for all subsets of W. Suppose there exists a σ'' for which $\sigma \dot{\sqsubset} \sigma''$ holds. For a σ'' to satisfy the condition $\sigma'' \vDash \psi \wedge val_Z(\sigma) = val_Z(\sigma'')$ all events that occur on σ have to occur in the same order on σ'' which is the case if $\sigma \dot{\sqsubseteq} \sigma''$ holds. The set W contains the event variables of the events that did not occur on σ and $val_W(\sigma)$ assigns *false* to all event variables in W. For $val_W(\sigma'')$ to be different from $val_W(\sigma)$ there has to be at least one event variable that is set to *true* by $val_W(\sigma'')$. This is only the case if an event that does not occur on σ occurs on σ''. Consequently, σ'' consists of all events that did occur on σ and at least one event that did not occur on σ which is true if $\sigma \dot{\sqsubset} \sigma''$ holds. $\sigma'' \nvDash_{CSL} \varphi$ holds if $\sigma'' \in \Sigma_B$ and is false if $\sigma'' \in \Sigma_G$. Hence, AC2(2) holds for σ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{\sqsubset} \sigma''$ holds. If $\sigma \dot{\sqsubset} \sigma''$ holds for some σ'' we need to modify ψ_σ as specified by Definition 8.

We will now show that AC3 is fulfilled by all traces in the candidate set $CFT(\varphi)$.

Theorem 4. *AC3 holds for all ψ_σ with $\sigma \in CFT(\varphi)$.*

Proof. AC(3) requires that no subset of the event order logic formula ψ satisfies AC1, AC2(1) and AC2(2). Suppose there exists a $\sigma''' \in \Sigma_C$ with $\sigma''' \subset \sigma$. We can partition $\mathcal{A}$ in $Z_{\sigma'''}$ and $W_{\sigma'''}$ such that $Z_{\sigma'''}$ consists of the variables of the events that occur on σ''' and $\psi_{\sigma'''}$ consists of the variables in $Z_{\sigma'''}$. For σ we partition $\mathcal{A}$ in Z_σ and W_σ such that Z_σ consists of the variables of the events that occur on σ and ψ_σ consists of the variables in Z_σ. Consequently, $Z_{\sigma'''} \subset Z_\sigma$ and $\psi_{\sigma'''} \subset \psi_\sigma$. If $\psi_{\sigma'''}$ satisfies AC1, AC2(1), AC2(2), then AC3 would be violated. But if such a σ'' would exist the minimality condition of the candidate set would be violated for σ and σ would not be part of the candidate set. Consequently, AC3 is fulfilled for all ψ_σ with $\sigma \in CFT(\varphi)$.

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

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

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 *Lr* indicates that the traffic light on the crossing is red, while the red light being off is denoted by the event *Lo*. The traffic light is red whenever a train is approaching and off when the train has left the crossing and no other train is approaching the crossing. It is possible that the traffic light fails (Lf) and in this case remains off although a train is approaching. A car will stop in front of the crossing if the gate is closed and the traffic light is red. Assume that the above described tests would identify the following event order logic formulas to be causal: $\mathrm{Gf} \wedge \mathrm{Tc} \wedge \mathrm{Cc}$ and $\mathrm{Lf} \wedge \mathrm{Tc} \wedge \mathrm{Cc}$. Due to the minimality property of the candidate set *CFT(φ)* the trace represented by the event order logic formula $\mathrm{Gf} \wedge \mathrm{Lf} \wedge \mathrm{Tc} \wedge \mathrm{Cc}$ would not be considered as being causal. 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.

Definition 13. *Combination Condition (CC1). Let $M = (S, Act, AP, L, \Sigma)$ a state-action trace model, and σ_1 and σ_2 state-action traces of M.*

CC1: *Let* $\sigma_1, \sigma_2, ...\sigma_k \in \mathrm{CFT}(\varphi)$ *traces and* $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_k}$ *the event order logic formulas representing them. A trace* σ *is added to* $\mathrm{CFT}(\varphi)$ *if for* $k \geq 2$ *traces in* $\mathrm{CFT}(\varphi)$ *it holds that* $(\sigma \vDash \psi_{\sigma_1}) \wedge (\sigma \vDash \psi_{\sigma_2}) \wedge ... \wedge (\sigma \vDash \psi_{\sigma_k})$ *and* $Z_\sigma = Z_{\sigma_1} \cap Z_{\sigma_2} \cap ... \cap Z_{\sigma_k}$.

We can now assign each trace in the candidate set the sum of the probability masses of the traces that it represents. This is done as follows: The probability of a trace σ_1 in *CFT(φ)* is the probability sum of all traces $\sigma' \in \Sigma_C$ for which σ_1 is the only subset in in *CFT(φ)*. The last condition is necessary in order to correctly assign the probabilities to traces which where added to the fault tree by test CC1.

Mapping to Fault Tree All traces 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 event* representing the top level event, that is connected to an *OR*-gate. The traces in the candidate set *CFT(φ)* will then be added as child nodes to the *OR*-gate as follows: A trace with a length of one and hence consisting of only one basic event is represented by the respective basic event. A trace 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 trace. If a (subset of a) trace 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 *(P)AND*-gates are the corresponding probabilities of all traces which 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 *(P)AND*-gate. The resulting intermediate events are then connected by an *OR*-gate that leads to the top event, representing the hazard. Since the trace probabilities are calculated for a trace 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.

Figure 2 shows the fault tree of the railroad crossing running example. For better readability we have omitted the order constraints of the *PAND*-gates. The top level event is represented by the *intermediate event* connected to the *OR*-gate. The *OR*-gate connects two *intermediate events* which are connected to the *PAND*-gates representing the event order logic formulas ψ_1 and ψ_2. The formula $\psi_1 = \mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \wedge \mathrm{Cc})) \wedge_< \neg \mathrm{Cl} \wedge_> \mathrm{Tc})$ is represented by the *PAND*-gate on the left side, and $\psi_2 = (\mathrm{Ta} \wedge (\mathrm{Ca} \wedge \mathrm{Cc})) \wedge_< \neg \mathrm{Cl} \wedge_> (\mathrm{Gc} \wedge \mathrm{Tc})$ by the *PAND*-gate on the right side.

3.4 Scalability and Complexity

[15] 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, in the general case computing causal relationships between variables

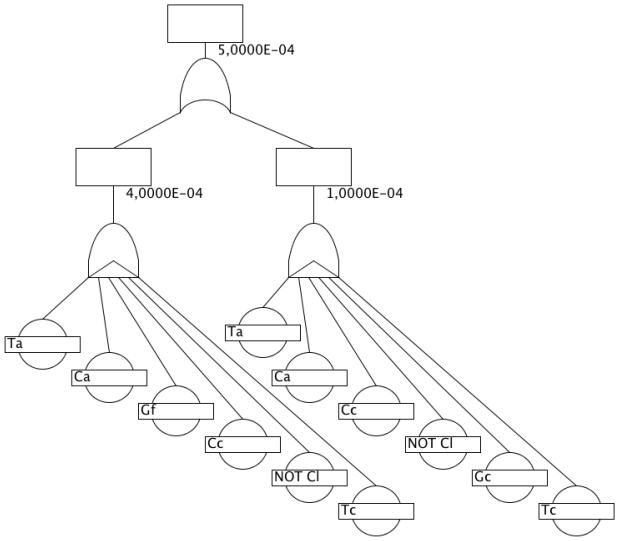

Fig. 2. Fault Tree of the Railroad Crossing Running Example

is NP-complete. Results in [16] show that causality can be computed in polynomial time if the causal graph over the variables which represent the events forms a directed causal tree. A directed causal tree consists of directed paths, containing the variables representing the events, leading to the root node representing the hazard or effect. Each trace in the counterexample is a directed path containing the variables representing the events leading to the hazard or effect. Consequently, a set of counterexamples resembles a directed causal tree and our algorithm can compute the causal process in polynomial time.

As we show in detail in [17], the complexity of our algorithm is cubic in the size of the counterexample. The 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 while the third is an industrial case study. 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 [18]. However, the case studies presented in this paper were directly modeled in the PRISM language. A comprehensive example how our causality computation method can be applied in a model-based setting can be found in [19]. We computed the counterexamples using our counterexample generation tool DiPro [6], which in turn

uses the PRISM model checker. All Experiments where performed on a PC with an Intel Core 2 Duo processor with 3.06 Ghz and 8 GBs of RAM.

4.1 Embedded Control System

This case study models an embedded control system based on the one presented in [20][3]. 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 $\mathcal{P}_{=?}(true\ U^{\leq t*3,600}\ down)$. We applied the XBF algorithm of DiPro in order to generate counterexamples for the property $\mathcal{P}_{=?}\ (true\ U^{\leq t*3,600}\ down)$. We computed the probability for the mission times t=10, t=100, and t=1000 hours. The resulting counterexample consists of 2024 paths. Figure 3 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 (InputPtoc-TransFail) or the failing of sensors / actuators (SensorFailure and SensorFailure (1) / ActuatorFailure and ActuatorFailure (1)). Figure 4 shows the memory

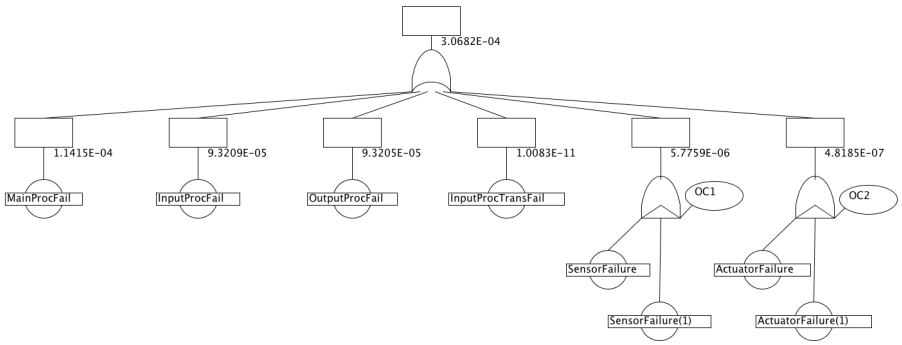

Fig. 3. Fault Tree of the Embedded Control System

and run time consumption of the counterexample and fault tree computation.

[3] The PRISM model of the embedded control system is part of the PRISM benchmark suite which can be obtained at http://www.prismmodelchecker.org/benchmarks/.

	Run time (sec. (min.))		Memory (MB)		Number of Paths	
t (h)	CX Comp.	FT Comp.	CX Comp.	FT Comp.	CX	FT
1	1 703 (≈ 28.38)	1.4	16	25	2024	6
10	2 327 (≈ 39)	1.3	16	25	2024	6
100	3 399 (≈ 56.60)	1.3	16	26	2024	6

Fig. 4. Experiment results of the embedded control system case study.

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. 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 is taken from [21]. 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 $P_{=?}[(\text{true})U^{<=t}$
$(unsafe)]$ where *unsafe* represents the above described unsafe state of the system and t represents the mission time. We applied the XBF algorithm of DiPro in order to generate counterexamples for the property with the mission times t=10, t=100, and t=1000. Figure 5 shows the fault tree generated from the counterexample for the formula $P_{=?}[(\text{true})U^{<=t}(unsafe)]$. 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. Figure

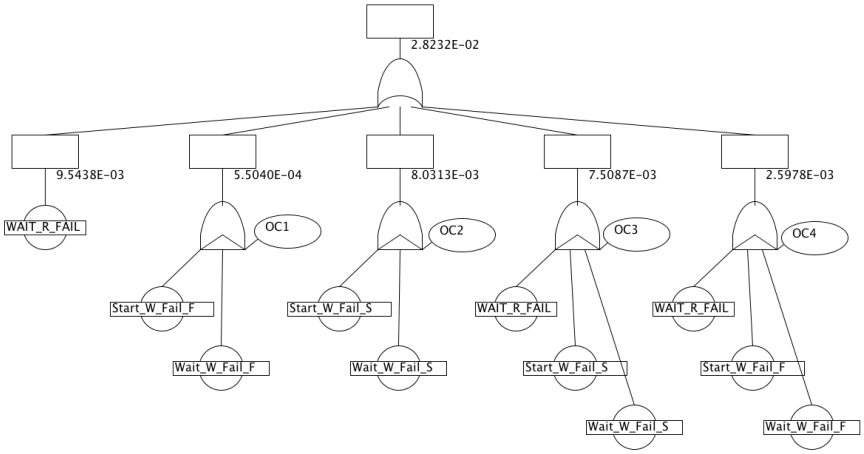

Fig. 5. Fault tree of the Train Odometer for T = 10

	Run time (sec. (min.))		Memory (MB)		Number of Paths	
t	CX Comp.	FT Comp.	CX Comp.	FT Comp.	CX	FT
10	433 (≈ 7.21)	7.9	78	122	108	5
100	582 (≈ 9.70)	8.0	78	116	108	5
1000	1 298 (≈ 21.63)	8.1	78	134	108	5

Fig. 6. Experiment results of the train odometer case study.

6 shows that the computational effort is dominated by the computation of the counterexample. 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. 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 Σ_G.

4.3 Airbag System

This case study is taken from [1] and models an industrial size airbag system. 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. 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. In CSL, this property can be expressed using the formula $\mathcal{P}_{=?}(noCrash\ U^{\leq t}\ AirbagIgnited)$. We applied the XBF algorithm of DiPro in order to generate counterexamples for the property with the mission times t=10, t=100, and t=1000. Figure 7 shows

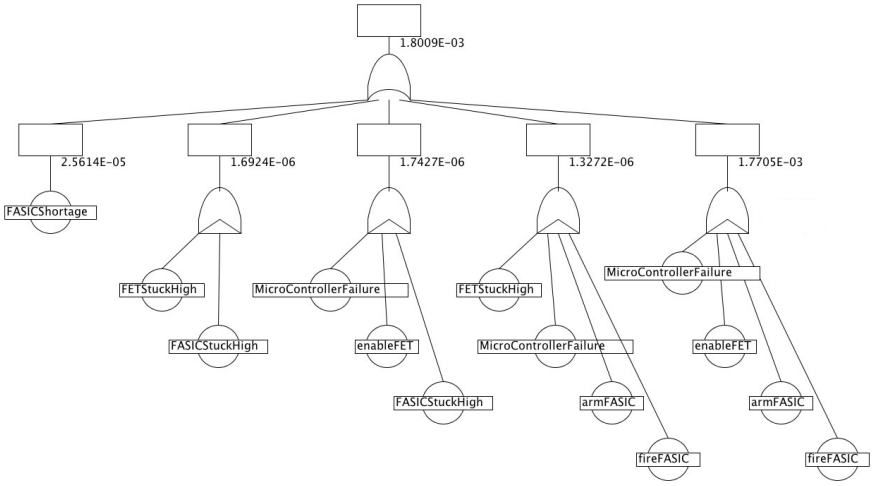

Fig. 7. Fault Tree of the Airbag System

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. For instance, there is only one single fault that can lead to an inadvertent deployment, namely *FASICShortage*. 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*. 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, along with the 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 computed by DiPro one would have to manually compare the order of the events in all 738 paths, which is a tedious and time consuming task. Figure 8 shows the memory and run time consumption of the counterexample and fault tree computation. 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. 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.

	Run time (sec. (min.))		Memory (MB)		Number of Paths	
t	CX Comp.	FT Comp.	CX Comp.	FT Comp.	CX	FT
10	1 147 (≈ 19.12)	1.3	29	27	738	5
100	1 148 (≈ 19.13)	1.3	29	27	738	5
1000	1 263 (≈ 21.05)	1.8	29	27	738	5

Fig. 8. Experiment results for the airbag case study.

5 Related Work

Work described in [14, 22] interprets fault trees in terms of temporal logic. 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 have been proposed, e.g. [23–25]. 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. Our work extends and improves on the approach of [21] in the following ways: We use a single set of system modeling and specification languages, namely PRISM and CSL. Whereas in the approach of [21] only minimal cut-sets are generated, we generate complete fault trees. Contrary to [21], we support PAND-gates and provide a justification for the causality model used. Work documented in [26] uses the Halpern and Pearl approach to explain counterexamples in functional CTL model checking by determining causality. However, this approach considers only functional counterexamples that consist of single execution sequences. Furthermore, it focuses on the causality of variable value-changes for the violation of CTL sub-formulas, whereas our approach identifies the events that lead to the variable value-changes. If for instance the CTL formula consists of only one boolean variable (e.g. *AirbagIgnited*), it is obvious that changing the value of the variable is causal for the property violation. The approach in [26] merely

identifies the variable value-change (e.g. setting *AirbagIgnited* to true) as cause, whereas our approach identifies the events that caused the value-change (e.g. FETStuckHigh and FASICStuckHigh). In [27] a formal framework for reasoning about contract violations is presented. In order to derive causality the notion of precedence established by Lamport clocks [28] is used. While this captures a partial order of the observed contract violations there is no evidence whether this partial order has an impact on causality or not. Work described in [29] establishes causality based on Lewis counterfactual reasoning by computing distance metrics between execution traces. The delta between the counterexample and the most similar good execution is identified as causal for the bad behavior. Due to the usage of Lewis counterfactual reasoning the approach is subject to the same limitations as the Lewis counterfactual reasoning. As an alternative to the event order logic that we defined we also investigated the usage of the interval logics [30] and [31]. We felt that in light of the relatively simple ordering constraints that we need to describe those logics are overtly expressive, and we hence decided to define our own tailored, relatively simple event order logic.

6 Conclusion

We presented a method and tool that automatically generates a fault tree from a probabilistic counterexample. 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 probabilistic 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 Structural 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 [32]. We are also interested in incorporating causality analysis directly into model checking algorithms. First results are presented in [33].

References

1. 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 the 6th Int. Conference on the Quantitative Evaluation of Systems (QEST 2009)*. IEEE Computer Society, 2009, pp. 299–308.
2. H. Aljazzar and S. Leue, "Debugging of Dependability Models Using Interactive Visualization of Counterexamples," in *Proc. of the 5th Int. Conference on Quantitative Evaluation of Systems (QEST 2008)*. IEEE Computer Society, 2008, pp. 189–198.

3. *Fault Tree Handbook*, U.S. Nuclear Regulatory Commission, 1981, nUREG-0492.
4. D. Lewis, *Counterfactuals*. Wiley-Blackwell, 2001.
5. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A Tool for Automatic Verification of Probabilistic Systems," in *Proc. of the 12th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2006)*, ser. LNCS, vol. 3920. Springer, 2006, pp. 441–444.
6. H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov, "Dipro - a tool for probabilistic counterexample generation," in *Model Checking Software - Pro. of the 18th Int. SPIN Workshop (SPIN 2011)*, ser. LNCS, vol. 6823. Springer, 2011, pp. 183–187.
7. J. Halpern and J. Pearl, "Causes and explanations: A structural-model approach. Part I: Causes," *The British Journal for the Philosophy of Science*, vol. 56, no. 4, pp. 889–911, 2005.
8. M. Kuntz, F. Leitner-Fischer, and S. Leue, "From probabilistic counterexamples via causality to fault trees," in *Proc. of Computer Safety, Reliability, and Security - 30th Int. Conference (SAFECOMP 2011)*, ser. LNCS, vol. 6894. Springer, 2011, pp. 71–84.
9. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, "Model-checking algorithms for continuous-time Markov chains," *IEEE Trans. on Software Engineering*, vol. 29, no. 6, pp. 524–541, 2003.
10. H. Aljazzar and S. Leue, "Directed explicit state-space search in the generation of counterexamples for stochastic model checking," *IEEE Trans. on Software Engineering*, vol. 36, no. 1, pp. 37–60, 2009.
11. T. Han, J.-P. Katoen, and B. Damman, "Counterexample generation in probabilistic model checking," *IEEE Trans. on Software Engineering*, vol. 35, no. 2, pp. 241–257, 2009.
12. J. Collins, Ed., *Causation and Counterfactuals*. MIT Press, 2004.
13. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.
14. G. Schellhorn, A. Thums, and W. Reif, "Formal fault tree semantics," in *Proc. of IDPT 2002*. Society for Design and Process Science, 2002.
15. T. Eiter and T. Lukasiewicz, "Complexity results for structure-based causality," *Artificial Intelligence*, vol. 142, no. 1, pp. 53–89, 2002.
16. ——, "Causes and explanations in the structural-model approach: Tractable cases," *Artificial Intelligence*, no. 6-7, pp. 542–580, 2006.
17. M. Kuntz, F. Leitner-Fischer, and S. Leue, "From probabilistic counterexamples via causality to fault trees," Chair for Software Engineering, University of Konstanz, Technical Report soft-11-02, 2011, uRL: http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-11-02.pdf.
18. F. Leitner-Fischer and S. Leue, "QuantUM: Quantitative safety analysis of UML models," in *Proc. of 9th Workshop on Quantitative Aspects of Programming Languages*, ser. EPTCS, vol. 57, 2011, pp. 16–30. [Online]. Available: http://www.inf.uni-konstanz.de/soft/research/publications/pdf/qapl2011.pdf
19. A. Beer, U. Kühne, F. Leitner-Fischer, S. Leue, and R. Prem, "Analysis of an airport surveillance radar using the quantum approach," Chair for Software Engineering, University of Konstanz, Technical Report soft-12-01, 2012, available from http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-12-01.pdf.
20. J. Muppala, G. Ciardo, and K. Trivedi, "Stochastic reward nets for reliability prediction," *Communications in Reliability, Maintainability and Serviceability*, vol. 1, no. 2, pp. 9–20, July 1994.

21. E. Böde, T. Peikenkamp, J. Rakow, and S. Wischmeyer, "Model Based Importance Analysis for Minimal Cut Sets," in *Proc. of the 6th Int. Symposium on Automated Technology for Verification and Analysis*, ser. LNCS, vol. 5311. Springer, 2008, pp. 303–317.
22. M. Bozzano, A. Cimatti, and F. Tapparo, "Symbolic Fault Tree Analysis for Reactive Systems," in *Proc. of the 5th Int. Symposium on Automated Technology for Verification and Analysis (ATVA 2007)*, ser. LNCS, vol. 4762. Springer, 2007, pp. 162–176.
23. G. Pai and J. Dugan, "Automatic synthesis of dynamic fault trees from UML system models," in *Proc. of the 13th Int.w Symposium on Software Reliability Engineering*. IEEE Computer Society, 2002, p. 243.
24. B. Chen, G. Avrunin, L. Clarke, and L. Osterweil, "Automatic Fault Tree Derivation From Little-Jil Process Definitions," in *Software Process Change*, ser. LNCS, vol. 3966. Springer, 2006, pp. 150–158.
25. 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 the 5th ACM Int. Conference on Embedded Software*. ACM, 2005, pp. 237–246.
26. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," in *Proc. of the 21st Int. Conference on Computer Aided Verification (CAV 2009)*, ser. LNCS. Springer, 2009, pp. 94–108.
27. G. Gössler, D. L. Métayer, and J.-B. Raclet, "Causality analysis in contract violation," in *Runtime Verification*, ser. LNCS, vol. 6418. Springer Verlag, 2010, pp. 270–284.
28. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563
29. A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *Int. Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 3, pp. 229–247, 2006.
30. R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt, "An interval logic for higher-level temporal reasoning," in *Proc. of the 2nd Annual ACM Symposium on Principles of Distributed Computing*. ACM, 1983, pp. 173–186.
31. L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. Ramakrishna, "A graphical interval logic for specifying concurrent systems," *ACM Trans. on Software Engineering and Methodology*, vol. 3, no. 2, pp. 131–165, 1994.
32. J. Dugan, S. Bavuso, and M. Boyd, "Dynamic Fault Tree Models for Fault Tolerant Computer Systems," *IEEE Trans. on Reliability*, vol. 41, no. 3, pp. 363–377, 1992.
33. F. Leitner-Fischer and S. Leue, "Causality checking for complex system models," in *Proc. of 14th Int. Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI2013)*, 2013.