Technical Report soft-13-01, Chair for Software Engineering, University of Konstanz, Copyright by the Authors 2013

On the Synergy of Probabilistic Causality Computation and Causality Checking

Florian Leitner-Fischer and Stefan Leue

University of Konstanz, Germany

Abstract. In recent work on the safety analysis of systems we have shown how causal relationships amongst events can be algorithmically inferred from probabilistic counterexamples and subsequently be mapped to fault trees. The resulting fault trees were significantly smaller and hence easier to understand than the corresponding probabilistic counterexample, but still contain all information needed to discern the causes for the occurrence of a hazard. More recently we have developed an approach called Causality Checking which is integrated into the state-space exploration algorithms used for qualitative model checking and which is capable of computing causality relationships on-the-fly. The causality checking approach outperforms the probabilistic causality computation in terms of run-time and memory consumption, but can not provide a probabilistic measure. In this paper we combine the strengths of both approaches and propose an approach where the causal events are computed using causality checking and the probability computation can be limited to the causal events. We demonstrate the increase in performance of our approach using several case studies.

1 Introduction

Model Checking [13] is an established technique for the verification of systems. For a formal model of the system and a formalized requirement the model checker automatically checks whether the model satisfies the requirement. In case the requirement is not satisfied, a trace from the initial system state into a state violating the requirement is produced by the model checker. This error trace is called a counterexample. Counterexamples can be used to retrace the steps of the system that lead to a particular requirement violating state, but they do not provide any insight into which event did cause the requirement violation. Consequently, debugging a system using counterexamples is a difficult iterative and hence time-consuming process.

In the case of probabilistic model checking [7] the debugging of the system becomes even more difficult. While in qualitative model checking a single trace often provides valuable information for the debugging of the system, a single trace is most often not sufficient to form a probabilistic counterexample [4,19] since the violation of a probabilistic property with a probability-bound can hardly ever be traced back to a single error trace. In almost all cases a set of error traces is needed to provide an accumulated probability mass that violates the probability-bound of the specified probabilistic property. With an increasing number of error traces that are needed to form the probabilistic counterexample, an increasing number of different error traces need to be manually retraced and interpreted in order to get insight into why the property was violated.

In recent work [24,28] we have developed two approaches that help to debug complex systems:

1. The *probabilistic causality computation* approach described in [24], where causal relationships of events are algorithmically inferred from probabilistic counterexamples and subsequently mapped to fault trees [33]. Fault trees are a method widely used in industry to visualize causal relationships. 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.
2. The *Causality Checking* approach [28], where the causality computation algorithm is integrated into the state-space exploration algorithms used for qualitative model checking. This algorithm is capable of computing the causality relationships on the fly.

The obvious advantage of the probabilistic causality computation approach over the causality checking approach is that it computes a quantitative measure, namely a probability, for a combination of causal events and hazards to occur. The probability of an event combination causing a property violation to occur is an information that is needed for the reliability and safety analysis of safety-critical systems. An important shortcoming of the probabilistic causality computation approach compared to the causality checking approach is that the causality computation requires a complete probabilistic counterexample consisting of all traces that violate the property. The high amount of run-time and memory that is needed to compute the probabilities of all traces in the probabilistic counterexample limits the scalability of the probabilistic causality computation approach.

The goal of this paper is to leverage the causality checking approach in order to improve the scalability of the probabilistic causality computation approach. The key idea is to first compute the causal events using the causality checking approach and to then limit the probability computation to the causal event combinations that have first been computed. Our proposed combined approach can be summarized by identifying the following steps:

– The probabilistic model is mapped to a qualitative model.
– The causality checking approach is applied to the qualitative model in order to compute the event combinations that are causal for the property violation.
– The information obtained through causality checking is mapped back to the probabilistic model. The probabilities for the different event combinations that are causal for the property violation to occur are computed using a probabilistic model checker.

The remainder of the paper is structured as follows: In Section 2 we briefly introduce probabilistic model checking, the PRISM language, and causality checking. We discuss the translation of probabilistic PRISM models to qualitative Promela models in Section 3. Section 4 is devoted to the translation of the information returned by the causality checker to the PRISM model and the probability computation of the causal events. In Section 5 we evaluate the usefulness of the proposed approach on several case studies. Related work is discussed throughout the paper and in Section 6. We conclude the paper and give an outlook on future research in Section 7.

2 Preliminaries

2.1 Probabilistic Model Checking

Probabilistic model checking [7] requires two inputs: a description of the system to be analyzed, typically given in some model checker specific modeling language, and a formal specification of quantitative properties of the system, related for example to its performance or reliability that are to be analyzed.

From the first of these inputs, a probabilistic model checker constructs the corresponding probabilistic model. This model is a probabilistic variant of a state-transition system, where each state represents a possible configuration of the system being modeled and each transition represents a possible evolution of the system from one configuration to another over time. The transitions are labeled with quantitative information specifying the probability and/or timing of the occurrence of the transition. The probabilistic models we use in this paper are continuous-time Markov chains (CTMCs) [23] where transitions are assigned positive, real values that are interpreted as rates of negative exponential distributions.

The quantitative properties of the system that are to be analyzed are specified using a variant of temporal logic. The temporal logic we use is Continuous Stochastic Logic (CSL) [1,6]. CSL is a stochastic variant of the Computation Tree Logic (CTL) [12] with state and path formulas based on [5].

We give a short introduction into CSL, for a more comprehensive description we refer to [6]. State formulas in CSL are interpreted over states of a CTMC, whereas path formulas are interpreted over paths in a CTMC. CSL extends CTL with two probabilistic operators that refer to the steady state and transient behavior of the model. The steady-state operator refers to the probability of residing in a particular set of states, specified by a state formula, in the long run, whereas the transient operator allows us to refer to the probability of the occurrence of particular paths in the CTMC. In order to express the time span of a certain path, the path operators until (U) and next (X) are extended with a parameter that specifies a time interval.

2.2 The PRISM Language

We present an overview of the input language of the PRISM model checker [25], for a precise definition of the semantics we refer to [21]. A PRISM model is composed of a number of *modules* which can interact with each other. A *module* contains a number of local variables. The values of these variables at any given time constitute the state of the *module*. The global state of the whole model is determined by the local state of all *modules*. The behavior of each module is described by a set of commands. A command takes the form: "[*action_label*] $guard \rightarrow rate_1 : update_1 \& ... \& \ update_n;$". The *guard* is a predicate over all variables in the model. The *update* commands describe a transition which the module can make if the *guard* is true. A transition is specified by giving the new values of the variables in the *module*, possibly as a function of other variables. A *rate* is assigned to each transition. The *action_label* is used for synchronizing transitions of different modules. If two transitions are synchronized they can only be executed if the guards of both transitions evaluate to true. The rate of the resulting synchronized transition is the product of the two individual transitions. An example of a PRISM model is given in Listing 1.1. The module named *moduleA* contains two variables: *var1*, which is of type Boolean and is initially *false*, and *var2*, which is a numeric variable and has initially the value 0. If the guard $(\text{var2} < 2)$ evaluates to true, the update $(\text{var2}' = \text{var2} + 1)$ is executed with a rate of 0.8. If the guard $(\text{var2} = 2)$ evaluates to true, the update $(\text{var1}' = \mathit{true})$ is executed with a rate of 1.0.

```
module moduleA
   var1: bool init false;
   var2: [0..11] init 0;
   [Count] (var2 < 4) -> 0.8: ( var2'= var2 + 1);
   [End] (var2 = 4) -> 1.0: ( var1'= true);
endmodule
module moduleB
   var3: [0..2] init 0;
   [Count] (var3 < 2) -> 1.0: ( var3'= var3 + 1);
   [Count] (var3 = 2) -> 1.0: ( var3'= 0);
endmodule
```

Listing 1.1. A module in the PRISM language.

The transitions with the action label *Count* of the modules moduleA and moduleB are synchronized. If the guard of the transition of *moduleA* labeled with *Count* evaluates to true and one of the guards of the transitions of *moduleB* labeled with *Count* evaluates to true, then the transition of moduleA will be executed simultaneously with the transition of moduleB for which the guard evaluates to true. If the guard of the transition of *moduleA* labeled with *Count* evaluates to true and both of the guards of the transitions of *moduleB* evaluate to true, one of the transitions of *moduleB* will be selected by a stochastic race and executed simultaneously with the transition of *moduleA*. If only the guard of the transition in *moduleA* labeled with *Count* evaluates to true, or only the guards of one or both of the transition in *moduleB* labeled with *Count* evaluate to true no transition will be executed.

2.3 Railroad Crossing Example

In this paper we will use the example of a railroad crossing for illustrative purposes. In this example a train can approach the crossing (Ta), enter the crossing (Tc) and finally leave the crossing (Tl). Whenever a train is approaching, the gate should close (Gc) and 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 enters the crossing (Cc) if the gate is open and finally leaves the crossing (Cl). The state of the railroad crossing in which both the car and the train are in the crossing at the same time is considered a hazardous and undesired state.

2.4 Causality Reasoning

The probabilistic causality computation approach and the causality checking approach are based on an adoption of the *structural equation model (SEM)* by Halpern and Pearl [18]. The SEM is an extension of the *counterfactual* reasoning approach and the *alternative world* semantics by Lewis [29,14]. The "naïve" 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. In our setting possible system execution traces represent the alternative worlds. The SEM introduces the notion of causes being logical combinations of events as well as a distinction of relevant and irrelevant causes. In the SEM events are represented by variable values and the minimal number of causal variable valuation combinations is determined. In our precursory work [24,28], we extended the SEM by considering the order of the occurrences of events as possible causal factors. In order to be able to reason about event orderings we defined a temporal logic called *event order logic* (EOL).

We will now give a brief overview of the EOL as originally defined in [28]. The EOL allows one to connect variables representing the occurrence of events with the boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events we introduced the ordered conjunction operator $\dot\wedge$. The formula $a \mathbin{\dot\wedge} b$ with events a and b is satisfied if and only if events a and b occur in a trace and a occurs before b. In addition to the $\dot\wedge$ operator we introduced the interval operators $\dot\wedge_{[}$, $\dot\wedge_{]}$, and $\dot\wedge_{<}\ \phi\ \dot\wedge_{>}$, which define an interval in which an event has to hold in all states. These interval operators are necessary to express the causal non-occurrence of events.

Definition 1. *Syntax of Event Order Logic (EOL). Simple EOL formulas over a set $\mathcal{A}$ of event variables are formed according to the following grammar:*

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

where $a \in \mathcal{A}$ and ϕ, ϕ_1 and ϕ_2 are simple EOL formulas. Complex EOL formulas are formed according to the following grammar:

$$\psi ::= \phi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \mathbin{\dot\wedge} \psi_2 \mid \psi \mathbin{\dot\wedge_{[}} \phi \mid \phi \mathbin{\dot\wedge_{]}} \psi \mid \psi_1 \mathbin{\dot\wedge_{<}} \phi \mathbin{\dot\wedge_{>}} \psi_2$$

where ϕ is a simple EOL formula and ψ_1 and ψ_2 are complex EOL formulas. Note that the $\neg$ operator binds more tightly than the $\dot\wedge$, $\dot\wedge_{[}$, $\dot\wedge_{]}$, and $\dot\wedge_{<}\ \phi\ \dot\wedge_{>}$, operators and those bind more tightly than the $\vee$ and $\wedge$ operator.

The formal semantics of this logic is defined over execution traces. Notice that the $\dot\wedge$, $\dot\wedge_{[}$, $\dot\wedge_{]}$, and $\dot\wedge_{<} \phi \dot\wedge_{>}$ operators are linear temporal logic operators and that the execution trace σ is akin to a linearly ordered Kripke structure.

Definition 2. *Semantics of Event Order Logic (EOL). Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, let ϕ, ϕ_1, ϕ_2 simple EOL formulas, let ψ, ψ_1, ψ_2 complex EOL 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 $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ a finite execution trace of T and $\sigma[i..r] = s_i, \alpha_{i+1}, s_{i+1}, \alpha_{i+2}, \ldots \alpha_r, s_r$ a partial trace. We define that an execution trace σ satisfies a formula ψ, written as $\sigma \vDash_e \psi$, as follows:*

$$s_j \vDash_e a_{\alpha_i} \text{ iff } s_{j-1} \xrightarrow{\alpha_i} s_j$$
$$s_j \vDash_e \neg\phi \text{ iff not } s_j \vDash_e \phi$$
$$\sigma[i..r] \vDash_e \phi \text{ iff } \exists j : i \leq j \leq r \;.\; s_j \vDash_e \phi$$
$$\sigma \vDash_e \psi \text{ iff } \sigma[0..n] \vDash_e \psi, \textit{ where } n \textit{ is the length of } \sigma.$$
$$\sigma[i..r] \vDash_e \phi_1 \wedge \phi_2 \text{ iff } \sigma[i..r] \vDash_e \phi_1 \textit{ and } \sigma[i..r] \vDash_e \phi_2$$
$$\sigma[i..r] \vDash_e \phi_1 \vee \phi_2 \text{ iff } \sigma[i..r] \vDash_e \phi_1 \textit{ or } \sigma[i..r] \vDash_e \phi_2$$
$$\sigma[i..r] \vDash_e \psi_1 \wedge \psi_2 \text{ iff } \sigma[i..r] \vDash_e \psi_1 \textit{ and } \sigma[i..r] \vDash_e \psi_2$$
$$\sigma[i..r] \vDash_e \psi_1 \vee \psi_2 \text{ iff } \sigma[i..r] \vDash_e \psi_1 \textit{ or } \sigma[i..r] \vDash_e \psi_2$$
$$\sigma[i..r] \vDash_e \psi_1 \dot\wedge \psi_2 \text{ iff } \exists j, k : i \leq j < k \leq r \;.\; \sigma[i..j] \vDash_e \psi_1 \textit{ and } \sigma[k..r] \vDash_e \psi_2$$
$$\sigma[i..r] \vDash_e \psi \dot\wedge_{[} \phi \text{ iff } (\exists j : i \leq j \leq r \;.\; \sigma[i..j] \vDash_e \psi \textit{ and } (\forall k : j \leq k \leq r \;.\; \sigma[k..k] \vDash_e \phi))$$
$$\sigma[i..r] \vDash_e \phi \dot\wedge_{]} \psi \text{ iff } (\exists j : i \leq j \leq r \;.\; \sigma[j..r] \vDash_e \psi \textit{ and } (\forall k : 0 \leq k \leq j \;.\; \sigma[k..k] \vDash_e \phi))$$
$$\sigma[i..r] \vDash_e \psi_1 \dot\wedge_{<} \phi \dot\wedge_{>} \psi_2 \text{ iff } (\exists j, k : i \leq j < k \leq r \;.\; \sigma[i..j] \vDash_e \psi_1 \textit{ and } \sigma[k..r] \vDash_e \psi_2$$
$$\textit{and } (\forall l : j \leq l \leq k \;.\; \sigma[l..l] \vDash_e \phi))$$

We define that the transition system T satisfies the formula ψ, written as $T \vDash_e \psi$, iff $\exists \sigma \in T \;.\; \sigma \vDash_e \psi$.

A system execution trace $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ induces an EOL formula $\psi_\sigma = a_{\alpha_1} \dot\wedge \ldots \dot\wedge a_{\alpha_n}$. For reasons of readability we omit the states in the execution traces from now on. For instance, the execution σ = Ta, Ca, Cc, Gc, Tc of the railroad example induces the EOL formula ψ_σ = Ta $\dot\wedge$ Ca $\dot\wedge$ Cc $\dot\wedge$ Gc $\dot\wedge$ Tc.

The adopted SEM defined in [24,28] can be used to decide whether the induced EOL formula ψ_σ of a execution traces on which the property is violated represent a causal combination of events. The conditions imposed by the adopted SEM for some ψ to be causal can be summarized as follows:

- AC1: This condition is the positive side of the counterfactual test. It checks whether there exists an execution trace σ that violates the property and satisfies the EOL formula ψ.
- AC2(1): This condition resembles the counterfactual test, where it is checked whether there exists an execution trace σ' where the order and occurrence of the events is different from ψ and the property is not violated.

- AC2(2): This condition says that for a ψ to be causal it can not be possible to add an event so that causality is voided. This test serves to reveal causal non-occurrence.
- AC3: This condition ensures minimality of the causal event combinations and requires that no sub-formula of ψ satisfies AC1 and AC2.
- OC1: This condition checks for all events in ψ whether the order in which they occur is causal or not.

For all executions where the property is violated the conditions imposed by the adopted SEM are checked. For instance, the safety property for the railroad crossing example is violated on the execution trace σ = Ta, Ca, Cc, Gc, Tc because the car is on the crossing when the gate closes and the train enters the crossing. Condition AC1 is fulfilled for $\psi_\sigma = \text{Ta} \dot\wedge \text{Ca} \dot\wedge \text{Cc} \dot\wedge \text{Gc} \dot\wedge \text{Tc}$ since σ exists and the property is violated. AC2(1) is fulfilled in this example since there exists the execution trace σ' = Ta, Ca, Gc, Tc where the occurrence and order of the events is different as specified in ψ_σ. For the AC2(2) test all good execution traces are needed to check whether there exists an event that can void the causality of ψ_σ. The condition AC2(2) reveals that there exists a good execution trace σ'' = Ta, Ca, Cc, Cl, Gc, Tc where the property is not violated because the car leaves the crossing before the gate closes (Gc) and the train enters the crossing (Tc). In other words, the non-occurrence of the event Cl between the event Cc and the events $(Gc \wedge Tc)$ is causal and its occurrence can void the causality of ψ_σ.

According to the procedures defined in [28] the causal non-occurrence of Cl is reflected by adding $\neg Cl$ to ψ_σ and we get $\psi_\sigma = \text{Ta} \dot\wedge \text{Ca} \dot\wedge \text{Gf} \dot\wedge \text{Cc} \dot\wedge_< \neg\text{Cl} \dot\wedge_> \text{Tc}$. AC3 is satisfied for ψ_σ because no subset of ψ_σ satisfies AC1 and AC2. Finally, OC1 checks for all events whether their order is causal or not. If their order is not causal the $\dot\wedge$ operator is replaced by the $\wedge$ operator. In our example, the order of the events Gf, Cc, ¬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 $\psi_\sigma = (\text{Ta} \wedge (\text{Ca} \dot\wedge \text{Cc})) \dot\wedge_< \neg\text{Cl} \dot\wedge_> (\text{Gc} \wedge \text{Tc})$. The disjunction of all $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_n}$ that satisfy the conditions AC1-AC3 and OC1 is the EOL formula describing all possible causes of the hazard. For the railroad crossing example the EOL formula returned by the causality checker is $\psi = (\text{Gf} \wedge ((\text{Ta} \wedge (\text{Ca} \dot\wedge \text{Cc})) \dot\wedge_< \neg\text{Cl} \dot\wedge_> \text{Tc})) \vee ((\text{Ta} \wedge (\text{Ca} \dot\wedge \text{Cc})) \dot\wedge_< \neg\text{Cl} \dot\wedge_> (\text{Gc} \wedge \text{Tc}))$.

Probabilistic Causality Computation [24]. In order to apply the probabilistic causality computation to a PRISM model first all traces in the counterexample and all good execution traces need to be computed using the DiPro tool [3]. The causality computation is subsequently performed as a post-processing step, where the conditions AC1-AC3 and OC1 are checked for all bad traces. Once the causality computation is completed, the probabilities of the execution traces in the probabilistic counterexample are assigned to the disjuncts of the EOL formula generated by the causality computation. The resulting EOL formula is then mapped onto a Fault Tree.

Causality Checking [28]. The algorithms used for causality checking are integrated into the state-space exploration algorithms used for model checking. The state-space of the model is traversed using breadth-first search or depth-first search. Whenever a bad trace violating the property or a good trace not entailing a property violation is found, this trace is added to a data-structure called sub-set graph. The conditions AC1-AC3 and OC1 are reduced to sub-execution test, thus the decision whether a combination of events is causal or not can be decided based on the position in the sub-set graph. Furthermore, this permits an on the fly decision whether a good trace needs to be stored for the AC2(2) test or whether it can be discarded.

2.5 Alternating Automata

In this paper we translate EOL formulas generated by the causality checker to alternating automata on finite words [11,34]. Alternating automata are a generalization of nondeterministic automata in which choices along a path can be marked existential, that is some branch has to reach an accepting state, or universal, which means that all branches have to reach an accepting state. We use the definition of alternating automata from [17] which differs from the definitions in [11,34] in the way that the automata are not defined with input symbols labeling the edges but with input symbols labeling the nodes instead.

Definition 3. *Alternating Automaton. An alternating automaton A is defined recursively as follows:*

$A ::= \epsilon_A$ *(empty automaton)*
$\mid \langle v, \delta, f \rangle$ *(conjunction of two automata)*
$\mid A_1 \vee A_2$ *(disjunction of two automata)*

where v is a state formula, δ is an alternating automaton expressing the next-state relation, and f indicates whether the node is accepting (denoted by +) or rejecting (−). We require the automaton be finite. The set of nodes of an automaton A, denoted by $\mathcal{N}(A)$ is formally defined as

$$\begin{aligned}
\mathcal{N}(\epsilon_A) &= \varnothing \\
\mathcal{N}(\langle v, \delta, f \rangle) &= \langle v, \delta, f \rangle \cup \mathcal{N}(\delta) \\
\mathcal{N}(A_1 \wedge A_2) &= \mathcal{N}(A_1) \cup \mathcal{N}(A_2) \\
\mathcal{N}(A_1 \vee A_2) &= \mathcal{N}(A_1) \cup \mathcal{N}(A_2)
\end{aligned}$$

A path through a nondeterministic automaton is a sequence of nodes. A "path" through an alternating automaton is, in general, a tree.

Definition 4. *Tree. A tree is defined recursively as follows:*

$T ::= \epsilon_T$ *(empty tree)*
$\mid T \cdot T$ *(composition)*
$\mid \langle \langle v, \delta, f \rangle, T \rangle$ *(single node with child tree)*

Definition 5. *Run of an Alternating Automaton. Given a finite sequence of states $\sigma = s_0, ..., s_{n-1}$ and an automaton A, a tree T is called a run of σ in A if* *one of the following holds:*

$$\begin{array}{lll}
A = \epsilon_A & \textit{and} & T = \epsilon_T \\
A = \langle v, \delta, f \rangle & \textit{and} & n > 1,\ T = \langle \langle v, \delta, f \rangle, T' \rangle,\ s_0 \vDash v \textit{ and } T' \textit{ is a run of } s_1, ..., s_{n-1} \\
 & & \textit{in } \delta, \textit{ or } n = 1,\ T = \langle \langle v, \delta, f \rangle, \epsilon_T \rangle \textit{ and } s_0 \vDash v \\
A = A_1 \wedge A_2 & \textit{and} & T = T_1 \cdot T_2, \textit{ where } T_1 \textit{ is a run of } A_1 \textit{ and } T_2 \textit{ is a run of } A_2 \\
A = A_1 \vee A_2 & \textit{and} & T \textit{ is a run of } A_1 \textit{ or } T \textit{ is a run of } A_2
\end{array}$$

Definition 6. *Accepting Run. A run is accepting if every path through the tree ends in an accepting node.*

For each alternating automaton A there exists a nondeterministic finite automaton A_n such that $L(A_n) = L(A)$, which was shown in [11,10,34].

3 Translating PRISM Models to Promela Models

Our goal is to compute the causal events using the causality checking approach and limit the probability computation to the causal events. To achieve this goal we need to translate the model given by a continuous-time Markov chain (CTMC) [23] specified in the PRISM language to a labeled transition system in the Promela language [22]. Due to space restrictions we can not introduce the Promela language here and refer to [22] for an in-depth introduction to Promela. Furthermore, the reachability property describing the hazard which is specified in Continuous Stochastic Logic (CSL) [1,6] needs to be translated into a formula in linear temporal logic [30]. The translation of the CSL formula to an LTL formula is straight forward: If the CSL formula is a state formula, then it is also an LTL formula. If the CSL formula is a path formula, then the path formula is an LTL formula if we replace a bounded-until operator inlcuded in the formula with an LTL until operator.

We base our translation of PRISM models to Promela models on the work in [32], but since no implementation of the described approach is available and the approach translates Markov Decision Processes specified in a PRISM model to a Promela model, we can not apply this approach directly. Furthermore, the in [32] proposed translation of synchronizing action labels to rendezvous channel chaining in Promela is not consistent with the PRISM semantics specified in [21]. Our translation algorithm maps the CTMC to a labeled transition system.

Definition 7. *Labeled Continuous-time Markov Chain (CTMC) [23]. A labeled Continuous-time Markov Chain C is a tuple* $(\mathcal{S}, s_0, \mathcal{R}, \mathcal{L})$*, where* $\mathcal{S}$ *is a finite set of states,* $s_0 \in \mathcal{S}$ *is the initial state,* $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ *is a transition rate matrix and* $\mathcal{L} : \mathcal{S} \to 2^{AP}$ *is a labeling function, which assigns to each state a subset of the set of atomic propositions* AP*.*

Definition 8. *Labeled Transition System [7]. A* transition system TS *is a tuple* $(S, Act, \to, I, AP, L)$ *where* S *is a finite set of* states*,* Act *is a finite set of* actions*,* $\to \subseteq S \times Act \times S$ *is a* transition relation*,* $I \subseteq S$ *is a set of* initial states*,* AP *is a set of* atomic propositions*, and* $L : S \to 2^{AP}$ *is a* labeling function*.*

Definition 9. *Transition System Induced by a CTMC. Let* $C = (\mathcal{S}, s_0, \mathcal{R}, \mathcal{L})$ *a CTMC then* $T = (S, Act, \rightarrow, I, AP, L)$ *is the transition system induced by* C *if:*

- *The set* S *of states in* T *is* $S = \mathcal{S}$.
- *The set* I *of initial states in* T *is* $I = \{s_0\}$.
- *For all pairs* $s, s' \in S$ *we add a transition to* $\rightarrow$ *and a corresponding action to* Act *if* $\mathcal{R}(s, s') > 0$.

We translate the induced transition system of the CTMC into the Promela language.

The implementation of the PRISM to Promela translation works on the syntax level of PRISM. PRISM *modules* are translated to *active proctypes* in Promela consisting of a *do*-block which contains the transitions. Transitions that are synchronized are translated according to the parallel composition semantics of PRISM [21]. All variables in the PRISM model are translated to global variables of the corresponding type in the Promela model. This is necessary, since otherwise it would not be possible to read variables from other proctypes as it is permitted in PRISM. Listing 1.2 shows the output of the PRISM to Promela translation of the PRISM code in Listing 1.1 from Section 2.2. The comments at the end of each transition are merely added to make the Promela model more readable but are not necessary for the translation.

Our approach requires that each command in the PRISM module is labeled with an action label representing the occurrence of an event. If a command of the PRISM model is not already labeled with an action label a unique action label is added to this command during the translation. This does not change the behavior of the PRISM model since the action label is unique and consequently is not synchronized with any other command.

```
bool var1 = false; byte var2 = 0; byte var3 = 0;
active proctype moduleA(){
  do
  :: atomic {((var3<2) && (var2<4)) -> var2=var2+1; var3=var3+1;}/*Count*/
  :: atomic {((var3==2) && (var2<4)) -> var2=var2+1; var3=0;}/*Count*/
  :: atomic {(var2==4) -> var1=true;}/*End*/
  od;}
active proctype moduleB(){
  do
  :: atomic {((var2<4) && (var3<2)) -> var3=var3+1; var2=var2+1;}/*Count*/
  :: atomic {((var2<4) && (var3==2)) -> var3=0; var2=var2+1;}/*Count*/
  od;}
```

Listing 1.2. Example Promela translation of the PRISM model from Section 2.2.

Listing 1.3 shows the PRISM code of the railroad crossing example. The Promela model generated by the PRISM to Promela translation is shown in Listing 1.4.

```
ctmc
module train
 s_train : [0..2] init 0;
 train_crossing : bool init false;
 [Ta] s_train = 0
       -> 0.01 : (s_train' = 1);
 [Tc] s_train = 1 & !gate_doUpdate
       -> 0.1 : (s_train' = 2) & (train_crossing' = true);
 [Tl] s_train = 2
       -> 0.1 : (s_train' = 0) & (train_crossing' = false);
endmodule
module car
 s_car : [0..2] init 0;
 car_crossing : bool init false;
 [Ca] s_car = 0
```

```
        -> 0.01 : (s_car' = 1);
 [Cc] s_car = 1 & gate_open
        -> 0.1 : (s_car' = 2)&(car_crossing' = true);
 [Cl] s_car = 2
        -> 0.1 : (s_car' = 0) & (car_crossing' = false);
endmodule
module gate
 gate_open : bool init true;
 gate_doUpdate : bool init false;
 gate_doUpdate : bool init failed;

 [Ta] !gate_failed
         -> 1.0 : (gate_doUpdate' = true);
 [Ta] gate_failed
         -> 1.0 : true;
 [Gc] gate_doUpdate & gate_open
         -> 0.9 : (gate_open' = false) & (gate_doUpdate' = false);
 [Gf] gate_doUpdate & gate_open
         -> 0.01 : (gate_open' = true)  & (gate_doUpdate' = false);
 [Gf] (true)
         -> 0.01 : (gate_open' = true)  & (gate_failed' = true);
 [Tl] !gate_failed
         -> 1.0 : (gate_doUpdate' = true);
 [Tl] gate_failed
         -> 1.0 : true;
 [gateOpening] gate_doUpdate & !gate_open
         -> 0.9 : (gate_open' = true)  & (gate_doUpdate' = false);
endmodule
```

Listing 1.3. PRISM model of the railroad example.

```
byte s_train = 0;
bool train_crossing = false;
byte s_car = 0;
bool car_crossing = false;
bool gate_open = true;
bool gate_doUpdate = false;
bool gate_failed = false;
active proctype train(){
  do
  :: atomic {((!gate_failed) && (s_train==0))
    -> s_train=1; gate_doUpdate=true;}/*Ta*/
  :: atomic {((gate_failed) && (s_train==0))
    -> s_train=1; gate_doUpdate=false;}/*Ta*/
  :: atomic {(s_train==1&&!gate_doUpdate)
    -> s_train=2; train_crossing=true;}/*Tc*/
  :: atomic {((!gate_failed) && (s_train==2))
    -> s_train=0; train_crossing=false; gate_doUpdate=true;}/*Tl*/
  :: atomic {((gate_failed) && (s_train==2))
    -> s_train=0; train_crossing=false; gate_doUpdate=false;}/*Tl*/
  od;}
active proctype car(){
  do
  :: atomic {(s_car==0)
    -> s_car=1;}/*car_aproaching*/
  :: atomic {(s_car==1&&gate_open)
    -> s_car=2; car_crossing=true;}/*car_crossing*/
  :: atomic {(s_car==2)
    -> s_car=0; car_crossing=false;}/*car_troughgate*/
  od;}
active proctype gate(){
  do:: atomic {((s_train==0) && (!gate_failed))
    -> gate_doUpdate=true; s_train=1;}/*Ta*/
  :: atomic {((s_train==0) && (gate_failed))
    -> gate_doUpdate=false; s_train=1;}/*Ta*/
  :: atomic {(gate_doUpdate&&gate_open)
    -> gate_open=false; gate_doUpdate=false;}/*Gc*/
  :: atomic {(gate_doUpdate&&gate_open)
    -> gate_open=true; gate_doUpdate=false;}/*Gf*/
  :: atomic {((true))
    -> gate_open=true; gate_failed=true;}/*Gf*/
  :: atomic {((s_train==2) && (!gate_failed))
    -> gate_doUpdate=true; s_train=0; train_crossing=false;}/*Tl*/
  :: atomic {((s_train==2) && (gate_failed))
    -> gate_doUpdate=false; s_train=0; train_crossing=false;}/*Tl*/
  :: atomic {(gate_doUpdate&&!gate_open)
    -> gate_open=true; gate_doUpdate=false;}/*Go*/
  od;}
```

Listing 1.4. Promela model of the railroad example.

Now that we can translate the PRISM model to a Promela model we can apply the qualitative causality checking approach. How the results of the qualitative causality checking can be mapped back to the PRISM model and used for probability computation is discussed in Section 4.

4 Computing Probabilities for Causal Events

For the railroad crossing example from Section 2.4 the EOL formula returned by the causality checker is $\psi = (\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> \mathrm{Tc})) \vee ((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> (\mathrm{Gc} \wedge \mathrm{Tc}))$. Intuitively, each disjunct of this formula represents a class of execution traces on which the events specified by the EOL formula cause the violation of the property.

In the rail road crossing example there are two classes of execution traces on which the hazard occurs.

1. If the gate fails (Gf) at some point of the execution and a train (Ta) and a car (Ca) are approaching this results in a hazardous situation if the car is on the crossing (Cc) and does not leave the crossing (Cl) before the train (Tc) enters the crossing $(\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> \mathrm{Tc}))$.
2. If a train (Ta) and a car (Ca) are approaching but the gate closes (Gc) when the car (Cc) is already on the railway crossing and is not able to leave (Cl) before the gate is closing and the train is crossing (Tc), this also corresponds to a hazardous situation $((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> (\mathrm{Gc} \wedge \mathrm{Tc}))$.

For instance, the execution traces $\sigma = \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Tc}$ and $\sigma' = \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gc}, \mathrm{Tc}, \mathrm{Tl}, \mathrm{Go}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Tc}$ are traces that belong to the first class of traces. The trace $\sigma'' = \mathrm{Ca}, \mathrm{Ta}, \mathrm{Cc}, \mathrm{Gc}, \mathrm{Tc}$ is an example for a trace in the second class.

We now formalize the observation that each disjunct of the EOL formula represents a class of traces by the notion of causality classes.

Definition 10. *Causality Class. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system and $\sigma = s_0, \alpha_1, s_1, \alpha_2, \dots \alpha_n, s_n$ a finite execution trace of T. The set Σ_B is the set of traces for which the property is violated.*

The causality classes $CC_1, ..., CC_n$ defined by the disjuncts of the EOL formula $\psi = \psi_1 \vee ... \vee \psi_n$ decompose the set Σ_B into sets $\Sigma_{B_{\psi_1}}, ..., \Sigma_{B_{\psi_n}}$ with $\Sigma_{B_{\psi_1}} \cup ... \cup \Sigma_{B_{\psi_n}} = \Sigma_B$.

Note that it can be the case that $\sigma \in \Sigma_{B_{\psi_1}} \wedge \sigma \in \Sigma_{B_{\psi_2}}$ if $\sigma \vDash_e \psi_1 \wedge \sigma \vDash_e \psi_2$.

All causal information that is needed in order to debug the system is represented by the causality classes. We can leverage this fact and compute the probability sum of all traces represented by a causality class instead of computing the probability of all traces belonging to this class individually. This means that the number of probabilistic model checking runs is reduced to the number of causality classes instead of the number of traces in the counterexample.

We will now show how the probability sum of all traces represented by a causality class can be computed using the PRISM model checker [25]. In order to compute the probability of all traces represented by a causality class we translate the EOL formula representing the causality class to an automaton which accepts exactly those execution traces that are represented by the corresponding causality class. Subsequently we show how we can synchronize the execution of this automaton with a PRISM model, such that the probability of all sequences which are accepted by the automaton is the probability sum of all traces represented by the corresponding causality class.

Note that since causality checking is limited to reachability properties a non-deterministic finite automaton (NFA) is sufficient to represent the finite execution traces represented by the causality class [7]. Since all orders of the events characterizing the causality class need to be considered, the size of the resulting NFA can be exponential in the size of the formula. To prevent this we use alternating automata on finite words [11,34] as defined in Section 2.5.

Given an EOL formula ψ we can construct an alternating automaton $A(\psi)$ such that $L(A(\psi)) = L(\psi)$. The construction of the automaton follows the structure of the formula.

Definition 11. *Alternating Automaton for an EOL formula. Let ψ an EOL formula that is built over the set of event variables $a \in \mathcal{A}$. The automaton $A(\psi)$ for the EOL formula ψ can be constructed following the structure of the formula as follows: For an event variable a: $A(a) = \langle a, \epsilon_A, + \rangle$, and for EOL formulas ψ_1, ψ_2 and ϕ_1:*

$$\begin{aligned}
A(\psi_1 \wedge \psi_2) &= A(\psi_1) \wedge A(\psi_2)\\
A(\psi_1 \vee \psi_2) &= A(\psi_1) \vee A(\psi_2)\\
A(\psi_1 \dot{\wedge} \psi_2) &= \langle true, A(\psi_1 \dot{\wedge} \psi_2), - \rangle \vee A_1 \quad \textit{where } A_1 = A(\psi_1) \wedge A_2\\
&\ \textit{and } A_2 = \langle true, A_2, - \rangle \vee A(\psi_2)\\
A(\phi_1 \dot{\wedge}_{]} \psi_1) &= A(\psi_1) \vee (\langle true, A(\phi_1 \dot{\wedge}_{]} \psi_1), - \rangle \wedge A(\phi_1))\\
A(\psi_1 \dot{\wedge}_{<} \phi_1 \dot{\wedge}_{>} \psi_2) &= \langle true, A(\psi_1 \dot{\wedge}_{<} \phi_1 \dot{\wedge}_{>} \psi_2), - \rangle \vee (A(\psi_1)\\
&\ \wedge(\langle true, A(\psi_1 \dot{\wedge}_{<} \phi_1 \dot{\wedge}_{>} \psi_2), - \rangle \vee \langle true, A(\phi_1 \dot{\wedge}_{]} \psi_2), - \rangle))
\end{aligned}$$

Note that since we consider only reachability properties, it can not be the case that an event voiding causality appears at the end of an execution trace. The EOL operator $\dot{\wedge}_{[}$ can hence not be added to an EOL formula as a consequence of AC2(2) and consequently we do not specify a translation rule for this operator. Notice that the only way for a $\neg$ operator to be added to an EOL formula by the causality checking algorithm is when the non-occurrence of the negated event in the specified interval is causal. To illustrate the proposed translation consider that for the EOL formula $\psi = (\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_{<} \neg\mathrm{Cl} \dot{\wedge}_{>} (\mathrm{Gc} \wedge \mathrm{Tc})$ of the railroad crossing example the first application of the recursive definition creates the following rewriting: $A(\psi) = \langle true, A((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_{<} \neg\mathrm{Cl} \dot{\wedge}_{>} (\mathrm{Gc} \wedge \mathrm{Tc})), - \rangle \vee (A((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc}))) \wedge (\langle true, A((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_{<} \neg\mathrm{Cl} \dot{\wedge}_{>} (\mathrm{Gc} \wedge \mathrm{Tc})), - \rangle \vee \langle true, A(\neg\mathrm{Cl} \dot{\wedge}_{]} (\mathrm{Gc} \wedge \mathrm{Tc})), - \rangle))$.

In order to compute the probability of a causality class we need to translate the corresponding alternating automaton into the PRISM language and synchronize it with the PRISM model.

Each action label in the PRISM model corresponds to an event variable in the set $\mathcal{A}$ over which the EOL formulas were built. As a consequence each alternating automaton accepts a sequence of PRISM action labels.

We will now define translation rules from alternating automata to PRISM modules. We call a PRISM module that was generated from an alternating automaton *causality class module*. The transitions of the causality class modules are synchronized with the corresponding transitions of the PRISM model. The transition rates of the causality class modules are set to 1.0, as a consequence, the transitions synchronizing with the causality class modules define the rate for the synchronized transition. In Listing 1.5 we present the pseudo-code of the algorithm that generates a causality class module from an alternating automaton representing an EOL formula.

The key idea is that for each event we add a boolean variable representing the occurrence of the event and a transition labeled with the action label of the event. The order constraints specified by the EOL formula are encoded by guards. Synchronized transitions can only be executed if for each other module containing transitions with the same action label the guard of at least one transition per module evaluates to true. It might hence be the case that the causality class module prevents the execution of transitions in the PRISM model with which the causality class module is synchronized. Since this would change the behavior of the PRISM model and affect the probability mass distribution we add for each transition of the causality class module for which the guard is not always true a transition with the negated guard and without updates.

We also add a PRISM *formula* acc_ψ for each sub-automaton which is true whenever the corresponding sub-automaton is accepting the input word. Those formulas are used to construct a CSL formula of the form $P_{=?}[(true)U(\text{acc\_}\psi)]$ for each causality class. The CSL formulas can then be used to compute the probability of all possible sequences that are accepted by the causality class module, which is the probability sum of all traces that are represented by the causality class. Since it its possible that a trace belongs to more than one causality class, we add an additional CSL formula that computes the probability of all traces that are only in the causality class defined by ψ. This CSL formula has the form of $P_{=?}[(true)U(\text{acc\_}\psi)\&!(\text{acc\_}\psi_i|...|\text{acc\_}\psi_j))]$, where $\text{acc\_}\psi_i|...|\text{acc\_}\psi_j$ are the formulas of all causality classes except ψ.

```
global var var_def = "", trans = "", formulas = "";
function EOL_TO_PRISM(A(ψ)){
  PRISM_CODE(A(ψ),true)
  print "module ψ \n" + var_def +"\n"+ trans
        + " \n endmodule \n" + formulas; }
function PRISM_CODE(A(ψ), cond){
  IF A(ψ) = 'A(a)' THEN
    var_def += 's_ψ: bool init false;'
    IF cond = 'true' THEN
      trans += '[a] (cond) -> 1.0 : (s_ψ'=true);'
    ELSE
      trans += '[a] (cond) -> 1.0 : (s_ψ'=true);'
      trans += '[a] !(cond) -> 1.0 : true;'
    ENDIF
    formulas += 'formula acc_ψ = s_ψ;'
  ELSE IF A(ψ) = 'A(ψ1)∧A(ψ2)' THEN
     PRISM_CODE(A(ψ1), cond); PRISM_CODE(A(ψ2), cond);
     formulas += 'formula acc_ψ = acc_ψ1 & acc_ψ2;'
  ELSE IF A(ψ) = 'A(ψ1 ∧ ψ2)' THEN
     PRISM_CODE(A(ψ1), cond); PRISM_CODE(A(ψ2), cond);
     formulas += 'formula acc_ψ = acc_ψ1 & acc_ψ2;'
  ELSE IF A(ψ) = 'A(ψ1 ∨ ψ2)' THEN
     PRISM_CODE(A(ψ1), cond); PRISM_CODE(A(ψ2), cond)
     formulas += 'formula acc_ψ = acc_ψ1 | acc_ψ2;'
  ELSE IF A(ψ) = 'A(ψ1)∨A(ψ2)' THEN
     PRISM_CODE(A(ψ1), cond); PRISM_CODE(A(ψ2), cond);
     formulas += 'formula acc_ψ = acc_ψ1 | acc_ψ2;'
  ELSE IF A(ψ) = 'A(ψ1 ⩑ ψ2)' THEN
     PRISM_CODE(A(ψ1), cond); PRISM_CODE(A(ψ2), acc_ψ1);
     formulas += formula acc_ψ = acc_ψ2;
  ELSE IF A(ψ) = 'A(φ1 ⩑] ψ1)' THEN
    PRISM_CODE(A(¬φ1), cond); PRISM_CODE(A(ψ1), cond & !(acc_¬φ1));
    formulas += 'formula acc_ψ = acc_ψ1;'
  ELSE IF A(ψ) = 'A(ψ1 ⩑< φ1 ⩑> ψ2)' THEN
     PRISM_CODE(A(ψ1), cond); PRISM_CODE(A(¬φ1), acc_ψ1)
     PRISM_CODE(A(ψ2), (acc_ψ1 & !(acc_¬φ1))
     formulas += 'formula acc_ψ = acc_ψ2;'
    ENDIF }
```

Listing 1.5. Pseudo-code of the EOL to PRISM algorithm.

Listing 1.6 shows the PRISM code of the EOL formula $(\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> (\mathrm{Gc} \wedge \mathrm{Tc})$ of the railroad crossing example.

```
module train_cc_2
  s_Ta : bool init false; s_Ca : bool init false;
  s_Cc : bool init false; s_Cl : bool init false;
  s_Tc : bool init false; s_Gc : bool init false;
  [Ta] (true) -> 1.0 : (s_Ta'=true);
  [Ca] (true) -> 1.0 : (s_Ca'=true);
  [Cc] (acc_Ca) -> 1.0 : (s_Cc'=true);
  [Cc] !(acc_Ca) -> 1.0 : true;
  [Cl] (acc_Ta_Ca_Cc) -> 1.0 : (s_Cl'=true);
  [Cl] !(acc_Ta_Ca_Cc) -> 1.0 : true;
  [Gc] (acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : (s_Gc'=true);
  [Gc] !(acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : true;
  [Tc] (acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : (s_Tc'=true);
  [Tc] !(acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : true;
endmodule
formula acc_Ta = s_Ta; formula acc_Ca = s_Ca;
formula acc_Ca_Cc = s_Ca & s_Cc;
formula acc_Ta_Ca_Cc = acc_Ta & acc_Ca_Cc;
formula acc_Cl = s_Cl; formula acc_Gc = s_Gc;
formula acc_Tc = s_Tc;
formula acc_Gc_Tc = acc_Gc & acc_Tc;
formula acc_train_cc_2 = acc_Gc_Tc;
```

Listing 1.6. PRISM code of the EOL formula $(\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> (\mathrm{Gc} \wedge \mathrm{Tc})$.

In the railroad example the total probability of a state where both the train and the car are on the crossing is $\mathrm{p\_total} = 2.312 \cdot 10^{-4}$. The proposed combined approach returns for the causality class characterized by $\psi_1 = \mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> \mathrm{Tc})$ the total probability of $p_{\psi_1} = 4.386 \cdot 10^{-5}$ and the exclusive probability of $p_{\psi_1}\mathrm{\_excl} = 3.464 \cdot 10^{-5}$, and for the causality class characterized by $\psi_2 = (\mathrm{Ta} \wedge (\mathrm{Ca} \dot{\wedge} \mathrm{Cc})) \dot{\wedge}_< \neg\mathrm{Cl} \dot{\wedge}_> (\mathrm{Gc} \wedge \mathrm{Tc})$ the total probability of $p_{\psi_2} = 1.970 \cdot 10^{-4}$ and the exclusive probability of $p_{\psi_2}\mathrm{\_excl} = 1.914 \cdot 10^{-4}$. We use the EOL to fault tree mapping proposed in [24] to visualize this results as a fault tree. Figure 1 shows the fault tree generated for the railroad crossing example.

5 Experimental Evaluation

In order to evaluate the proposed *combined approach*, we have extended the SpinCause tool. SpinCause is based on the SpinJa toolset [15], a Java re-implementation of the explicit state model checker Spin [22]. The following experiments were performed on a PC with an Intel Xeon Processor (3.60 Ghz) and 144 GBs of RAM. We evaluate the combined approach on a case study from the PRISM benchmark suite [26] and two industrial case studies [2,8] for which the PRISM models where automatically generated by the QuantUM tool [27] from a higher-level architectural modeling language. The extended SpinCause tool and the PRISM models used in this paper can be obtained from `http://se.uni-konstanz.de/research1/tools/spincause`.

5.1 Case Studies

Embedded Control System [31]. The PRISM model of the embedded control system is part of the PRISM benchmark suite [26]. The system consists of a main

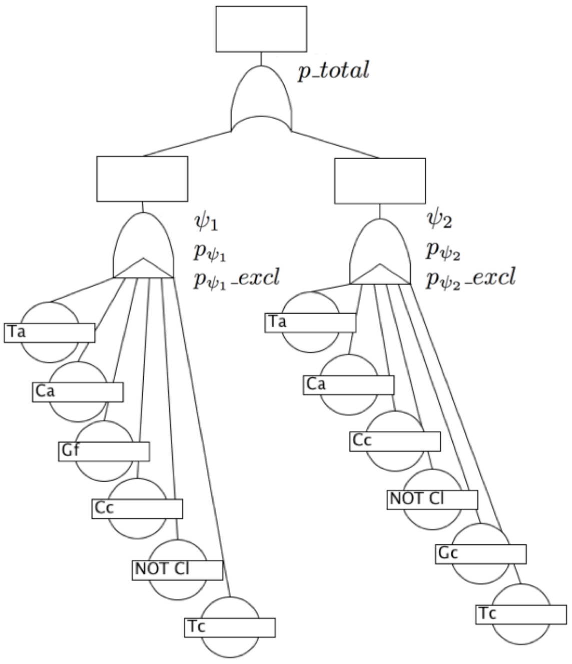

Fig. 1. Fault tree of the railroad crossing example.

processor, an input processor, an output processor, 3 sensors, and two actuators. Various failure modes can lead to a shutdown of the system. We are interested in computing the causal events for an event of the type "system shut down within one hour". Since one second is the basic time unit in our system one hour corresponds to a mission time of T=3,600 time units. The formalization of this property in CSL reads as $P_{=?}(true\ U^{\leq T}\ down)$. We set the constant *MAX_COUNT*, which represents the maximum number of processing failures that are tolerated by the main processor, to a value of 5.

Airbag System [2]. This case study models an industrial size airbag system. It contains an behavioral description of all system components that are involved in deciding whether a crash has occurred. It is a pivotal safety requirement that an airbag is never deployed if there is no crash situation. We are interested in computing the causal events for an inadvertent ignition of the airbag. In CSL, this property can be expressed using the formula $P_{=?}(noCrash\ U^{\leq T}\ AirbagIgnited)$. The causality checker returned 5 causality classes. The total probability for an inadvertent deployment of the airbag within T=100 computed by the combined approach is p_total = 0.228.

Train Odometer Controller [8]. The train odometer system consists of two independent sensors used to measure the speed and the position of a train. A monitor component continuously checks the status of both sensors. It reports failures of the sensors to other train components that have to disregard temporarily erroneous sensor data. 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. We are interested in computing the causal events for reaching an unsafe state of the system. This can be expressed by the CSL formula $P_{=?}[(\text{true})U^{<=T}(\mathit{unsafe})]$.

	Combined Approach		Probabilistic Causality Comp.	
	Run time (sec.)	Memory (MB)	Run time (sec.)	Memory (MB)
Embedded: States: 6,013 Transitions: 25,340				
T=10	3.06	19.27	2,003.00	409
T=3600	4.79	19.29	2,102.00	409
Airbag: States: 2,952 Transitions: 14,049				
T=10	10.88	52.44	682.00	154
T=1000	33.63	52.44	874.00	154
Train Odometer Controller: States: 117,222 Transitions: 66,262				
T=10	91.37	195.29	16,191.00	1,886
T=1000	2,572.74	195.29	44,356.00	1,886

Table 1. This table shows the experiment results with the combined approach and the probabilistic causality computation approach.

5.2 Discussion

As we would expect, for all case studies the total probability returned by the combined approach is equal to the probability returned for the respective probabilistic property by PRISM after a probabilistic model checking run. If we sum up the probabilities of the traces computed by DiPro for each causality class and only consider traces that belong to exactly one causality class, then the sum of the probability of each causality class is equal to the corresponding p_ψ_excl value of that causality class computed by the combined approach. If, on the other hand, we sum up the probabilities of of the traces computed by DiPro for each causality class and also consider the probability mass of traces that belong to more than one causality class, the the sum of each causality class is equal to the corresponding p_ψ value of that causality class computed by the combined approach. These observations make us confident that the combined approach computes correct probabilities.

Table 1 shows the run time and memory consumption of the combined approach and the probabilistic causality computation approach for each of the case studies. The combined approach consumes significantly less run time and memory than the probabilistic causality computation approach. This difference can be explained by the fact that for the probabilistic causality approach the probability of each traces in the counterexample needs to be computed individually, which requires a probabilistic model checking of a part of the model for each trace. The combined approach reduces the number of probabilistic model checking runs to the number of the computed causality classes. The run time of the combined approach increases with the mission time T because the time needed by the PRISM model checker to compute the probability for the different causality classes increases with an increasing T. The relatively low runtime that is needed by the combined approach for the embedded case study as compared to the other case studies can be explained by the relatively short length of the traces in the causality classes of the embedded case study.

6 Related Work

A translation from Markov decision processes (MDPs) into the PRISM language has been proposed in [32], but no implementation of the tool is publicly available. Furthermore, the proposed translation of synchronizing action labels to rendezvous channel chaining in Promela is not consistent with the PRISM semantics specified in [21].

In [9], a formalization of the semantics of dynamic fault trees (DFTs) [16] and a probabilistic analysis framework for DFTs based on interactive Markov chains [20] is presented. The approach in [9] takes the DFT as the only input. As a consequence, while this approach allows for a probabilistic analysis of the events in the DFT, there is no possibility to combine the analysis with a model containing the events of the DFT.

The approach of [8] computes minimal-cut sets, which are minimal combinations of events that are causal for a property violation, and their corresponding probabilities. Our approach extends and improves this approach by considering the event order as a causal factor. Work in [19] documents how probabilistic counterexamples for discrete-time Markov chains (DTMCs) can be represented by regular expressions. While the regular expressions define an equivalence class for some traces in the counterexample, it is possible that not all possible traces are represented by the regular expression and consequently not all causal event combinations are captured by the regular expression. In [4,35] probabilistic counterexamples are represented by identifying a portion of an analyzed Markov chain in which the probability to reach a safety-critical state exceeds the probability bound specified by an upper-bounded reachability property. The method proposed in this paper improves these approaches by identifying not only a portion of the Markov chain, but all event combinations and their corresponding order. Furthermore, the approach presented in [35] is applicable to DTMCs and MDPs, whereas our approach is applicable to CTMCs. In addition none of the approaches in [8,19,4,35] is able to reveal that the non-occurrence of an event is causal.

To the best of our knowledge there is no approach in the literature that combines qualitative causality reasoning with probabilistic causality computation.

7 Conclusion

We have discussed how the qualitative causality checking approach can be leveraged in order to improve the scalability of the probabilistic causality computation approach. Furthermore, we have proposed and implemented a mapping of CTMC models in the PRISM language to transition systems in the Promela language. In addition, we have shown how an EOL formula generated by the qualitative causality checking approach can be translated into an equivalent alternating automaton, and how the resulting alternating automaton can be translated to a causality class module in the PRISM language. The resulting causality class module can then be used to compute the probability sum of all traces represented by the causality class. We have demonstrated the performance increase of the proposed synergy approach compared to the probabilistic causality computation on several case studies from academia an industry.

In future work we plan to extend the combined approach to support DTMC and MDPs models.

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. Springer, 1996.
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*. IEEE Computer Society, 2009.
3. H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov. Dipro - a tool for probabilistic counterexample generation. In *Proceedings of the 18th International SPIN Workshop*, volume 6823 of *LNCS*, pages 183–187. Springer, 2011.
4. H. Aljazzar and S. Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. Soft. Eng.*, 2009.
5. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-Checking Continuous-Time Markov Chains. *ACM Trans. Comput. Logic*, 1(1):162–170, 2000.
6. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Soft. Eng.*, 2003.
7. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
8. 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.* Springer, 2008.
9. H. Boudali, P. Crouzen, and M. Stoelinga. A rigorous, compositional, and extensible framework for dynamic fault tree analysis. *Dependable and Secure Computing, IEEE Transactions on*, 7(2):128–143, 2010.
10. J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, 1980.
11. A. K. Chandra and L. J. Stockmeyer. Alternation. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 98–108. IEEE, 1976.
12. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

13. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking (3rd ed.).* The MIT Press, 2001.
14. J. Collins, editor. *Causation and Counterfactuals.* MIT Press, 2004.
15. M. de Jonge and T. Ruys. The spinja model checker. In *Model Checking Software.* Springer, 2010.
16. J. Dugan, S. Bavuso, and M. Boyd. Dynamic Fault Tree Models for Fault Tolerant Computer Systems. *IEEE Trans. Reliability*, 1992.
17. B. Finkbeiner and H. Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.
18. J. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Phil. of Science*, 2005.
19. T. Han, J.-P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. Softw. Eng.*, 2009.
20. H. Hermanns. *Interactive Markov chains: and the quest for quantified quality.* Springer-Verlag, 2002.
21. A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. The prism language - semantics. Available from URL `http://www.prismmodelchecker.org/doc/semantics.pdf`.
22. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addision–Wesley, 2003.
23. V. Kulkarni. *Modeling and analysis of stochastic systems.* Chapman & Hall/CRC, 1995.
24. M. Kuntz, F. Leitner-Fischer, and S. Leue. From probabilistic counterexamples via causality to fault trees. In *Proceedings of Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011.* Springer, 2011.
25. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
26. M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST'12)*, pages 203–204. IEEE CS Press, 2012.
27. F. Leitner-Fischer and S. Leue. QuantUM: Quantitative safety analysis of UML models. In *Proc. of the 9th Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, 2011.
28. 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.
29. D. Lewis. *Counterfactuals.* Wiley-Blackwell, 2001.
30. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., 1992.
31. 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.
32. C. Power and A. Miller. Prism2promela. In *Quantitative Evaluation of Systems, 2008. QEST'08. Fifth International Conference on*, pages 79–80. IEEE, 2008.
33. U.S. Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981.
34. M. Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency*, pages 238–266, 1996.

35. R. Wimmer, N. Jansen, E. Ábrahám, B. Becker, and J.-P. Katoen. Minimal critical subsystems for discrete-time markov models. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–314, 2012.