**Technical Report soft-12-02, Revision 2, Chair for Software Engineering, University of Konstanz, Copyright by the Authors 2012**

# Causality Checking for Complex System Models

Florian Leitner-Fischer and Stefan Leue

University of Konstanz, Germany

**Abstract.** We present an approach for the algorithmic computation of causalities in system models that we refer to as *causality checking*. We are basing our notion of causality on counterfactual reasoning, in particular using the structural equation model approach by Halpern and Pearl that we recently have extended to reason about computational models. In this paper we present a search-based on-the-fly approach that nicely integrates into finite state verification techniques, such as explicit-state model checking. We demonstrate the applicability of our approach using an industrial case study.

## 1 Introduction

With the increasing complexity of modern safety-critical systems, the need for model based engineering methods that both help in architecting such systems and to asses their safety and correctness becomes increasingly obvious. Due to the size of the systems traditional techniques like reviews [1] and testing, on the one hand, and manual fault tree analysis [2] or failure mode and effect analysis [3] on the other hand, can only be applied to limited parts of the system. The main reason for this limitation lies in the vast amount of time and resources that is consumed by manually executing those techniques. In order to be able to asses the correctness and safety of these systems in a comprehensive manner automated or, at least, computer-aided techniques are needed.

Model Checking [4] is an established technique for the automated analysis of system properties. If a model of the system and a formalized property is given to the model checker, it automatically checks whether it can find property violations. In case some property is violated, the model checker returns a counterexample, which consists of a system execution trace leading to the property violation. While a counterexample helps in retracing the system execution leading to the property violation, it does not identify causes of the property violation.

We present an approach based on explicit state space search towards the automated computation of causalities that we refer to as *causality checking*. Instead of returning just a single counterexample at the end of the model checking process, we compute causal events that lead to the violation of a desired system property. The notion of causality that we use is based on counterfactual reasoning [5, 6].

In precursory work [7] causality computation was performed as a postprocessing step on a set of probabilistic counterexamples. In addition we presented a mapping of the computed causality relationships between events to fault trees. For the causality computation all possible execution traces need to be computed and stored on disk prior to the causality checking. The current paper focuses on a extension of our causality model and an integration of the causality computation into standard state-space search as used by explicit-state model checkers. Consequently, it is no longer necessary to store all good and bad execution traces before performing the causality computation. We tailor the causality model from [7] so that it can be used for the analysis of concurrent system models described by transition systems. We also show how the causality checks can be mapped to sub- and superset comparisons of execution traces. The proposed algorithm for causality checking is an extension of the depth-first search and breadth-first search algorithms used for state-space exploration in explicit-state model checking. In keeping with standard practice in this domain we design our algorithms to work on-the-fly. To his end we propose a data-structure called subset graph that is used to store the counterexamples that are needed for causality checking. A further contribution of our current paper is an application of this approach to two case studies, one of them of industrial size, and a comparison of various search strategies.

The remainder of this paper is structured as follows. In Section 2 we discuss how causality relationships can be formally established within system models. The on-the-fly algorithm for causality computation and its integration in state-space exploration algorithms is presented in Section 3. In Section 4 we demonstrate the causality checking approach using two case studies. Related work is discussed throughout the paper and in Section 5. We conclude in Section 6.

## 2 Causality Reasoning in System Models

Our goal is to identify the events that cause the violation of a non-reachablitiy requirement. Such a violation could, for instance, represent a hazard or a potentially unsafe state of the system. We use the explicit state model checker SPIN [8] to check whether there are system executions that lead to such an undesired state. Before we explain in Section 2.2 how causality checking can be performed on system executions, we give the underlying formalization of the system model.

### 2.1 System Model

The systems that we wish to apply causality checking to are concurrent systems. For the formalization of the system model we follow the formalization of a model for concurrent computing systems proposed in [9]. The system model is given by a Transition System which is defined as follows:

**Definition 1.** *Transition System. A* transition system *$TS$ is a tuple* $(S, Act, \rightarrow, I, AP, L)$ *where $S$ is a finite set of* states*, $Act$ is a finite set of* actions*,*

$\rightarrow \subseteq S \times Act \times S$ *is a* transition relation*,* $I \subseteq S$ *is a set of* initial states*,* $AP$ *is a set of* atomic propositions*, and* $L : S \rightarrow 2^{AP}$ *is a* labeling function*.*

A Transition System defines a Kripke structure. Each state $s \in S$ is labeled with the set $L(s)$ of all atomic state propositions that are true in this state. The set $Act$ contains all actions that can trigger the system to transit from some state into a successor state.

The execution semantics of a transition system is defined as follows:

**Definition 2.** *Execution Trace of a Transition System. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *be a transition system. A finite execution* $\sigma$ *of* $T$ *is an alternating sequence of states* $s \in S$ *and actions* $\alpha \in Act$ *ending with a state.* $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ ...\ \alpha_n\ s_n$ *s.t.* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ *for all* $0 \leq i < n$*.*

The analysis aims at identifying the violation of functional safety requirements. Such a violation is also referred to as a hazard. We use linear time temporal logic (LTL) using its standard syntax and semantics as defined in [10] in order to specify hazards. Hazards imply the reachability of unsafe states and they hence belong to the class of reachability properties. Hence we only need to consider finite execution fragments [9]. Hazards fall within the class of safety properties in the commonly used classification scheme of safety and liveness properties. We use $T \vDash_l \varphi$ to express that the LTL formula $\varphi$ holds for the transition system $T$ and $\sigma \vDash_l \varphi$ respectively for execution traces.

We will demonstrate the presented definitions on a running example of a railroad crossing system. In the running example 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. This hazard can be characterized by the LTL formula $\varphi = \Box\neg(car\_crossing \land train\_crossing)$.

In the following we will use short-hand notation $\sigma =$ "$a_{\alpha_1}$,$a_{\alpha_2}$, ... , $a_{\alpha_n}$" for an execution trace $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ ...\ \alpha_n\ s_n$. The trace $\sigma =$ "Ta, Ca, Gf, Cc, Tc", for instance, is a trace of the railroad example where the train and the car are approaching the crossing (Ta, Ca), the gate fails to close (Gf), the car crosses the crossing (Cc) and finally the train crosses the crossing (Tc).

We can partition the set of all possible execution traces $\Sigma$ of a transition system $T$ into the set of "good" execution traces, denoted $\Sigma_G$, where the LTL formula is not violated and thus the hazard does not occur, and the set of "bad" execution traces, denoted $\Sigma_B$, where the LTL formula is violated and thus the hazard occurs. The elements of $\Sigma_B$ are also referred to as counterexamples in model checking. The trace $\sigma =$ "Ta, Ca, Gf, Cc, Tc" we already discussed above is a "bad" execution trace, since bot the car and the train are on the crossing at the same time and thus the LTL property is violated. An example for a "good" trace is $\sigma' =$ "$Ta, Ca, Gf, Cc, Cl, Tc$" where the car leaves the crossing (Cl) before the train is crossing (Tc) and consequently the train and the car are not on the crossing at the same time and the LTL formula is not violated.

**Definition 3.** *Good and Bad Execution Traces. Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system, let $\varphi$ an LTL formula over AP and $\Sigma$ that set of all possible finite executions of T. The set $\Sigma$ is divided into into the set of "good" execution traces $\Sigma_G$ and in the set of "bad" execution traces $\Sigma_B$ as follows: $\Sigma_G = \{\sigma \in \Sigma \mid \sigma \vDash_l \varphi\}$, $\Sigma_B = \{\sigma \in \Sigma \mid \sigma \nvDash_l \varphi\}$ and $\Sigma_G \cup \Sigma_B = \Sigma$ and $\Sigma_G \cap \Sigma_B = \varnothing$.*

## 2.2 Causality Reasoning

Our goal is to automatically identify those events that are causal for the violation of an LTL property. We assume that for a given execution trace $\sigma$ of a transition system $T$, $Act$ contains the events that we wish to reason about. For an LTL formula $\varphi$ specifying a safety requirement and an execution trace $\sigma$, the hazard described by the safety requirement occurs on $\sigma$ if and only if $\sigma \nvDash_l \varphi$ holds. Notice that since each transition is only labeled with one action, only one event can occur per transition. In order to be able to reason about the causality of events we have to formally capture the occurrence of events. We assume that there exists a set $\mathcal{A}$ of event variables that contains a boolean variable $a$ for each action $\alpha \in Act$ for some given transition system. The variable $a_{Ta}$ for instance represents the event train approaching the crossing. If multiple instances of one event type occur on one execution trace, for example the two train approaching events on "Ta,Gc,Tc,Tl,Go,Ta", the variables representing them are numbered according to their occurrence, for our example $a_{Ta_1}$ and $a_{Ta_2}$. In other words, the *i-th* occurrence of some action of type $\alpha$ will be represented by the boolean variable $a_{\alpha_i}$. In the following we also abbreviate the event variable $a_{Ta}$ by Ta.

**Definition 4.** *Events, Event Types and Event Variables. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system and $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ a finite execution trace of T. We define the following: each $\alpha \in Act$ defines an* event type *$\alpha$. $\alpha_i$ of $\sigma$ denotes the* i-th *occurrence of an event of the* event type *$\alpha$. The variable representing the occurrence of the event $\alpha_i$ is denoted by $a_{\alpha_i}$, and the set $\mathcal{A} = \{a_{\alpha_1}, ..., a_{\alpha_n}\}$ contains a boolean variable for each occurrence of an event.*

Event variables allow us to reason about the occurrence of single events, but since we want to reason about the combination of events, we need a formalism that allows us to express the occurrence of event combinations. In [7] we presented the event order logic (EOL) which allows one to connect event variables from $\mathcal{A}$ with the boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events we introduced the ordered conjunction operator $\curlywedge$. The formula $a \curlywedge b$ with $a, b \in \mathcal{A}$ is satisfied if and only if events $a$ and $b$ occur in a trace and $a$ occurs before $b$. We present here an amended version of the event order logic and further refine it in order to enable causality reasoning for concurrent system models specified by transition systems. In addition to the $\curlywedge$ operator we introduce the interval operators $\curlywedge_{[}$, $\curlywedge_{]}$, and $\curlywedge_{<} \phi \curlywedge_{>}$, which define an interval in which an event has to hold in all states. As we will see later, these interval operators are necessary to express the causal non-occurrence of events.

**Definition 5.** *Syntax of Event Order Logic (EOL). Simple event order logic formulas over the 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 $\phi$, $\phi_1$ and $\phi_2$ are simple event order logic formulas. Complex event order logic formulas are formed according to the following grammar:*

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

*where $\phi$ is a simple event order logic formula and $\psi_1$ and $\psi_2$ are complex event order logic 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 on 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 $\sigma$ is akin to a linearly ordered Kripke structure.

**Definition 6.** *Semantics of Event Order Logic (EOL). Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, let $\phi$, $\phi_1$, $\phi_2$ simple event order logic formulas, let $\psi$, $\psi_1$, $\psi_2$ complex event order logic formulas, and let $\mathcal{A}$ a set of event variables, with $a_{\alpha_i} \in \mathcal{A}$, over which $\phi$, $\phi_1$, $\phi_2$ are built. Let $\sigma = s_0, \alpha_1, s_1, \alpha_2, \dots \alpha_n, s_n$ a finite execution trace of $T$ and $\sigma[i..r] = s_i, \alpha_{i+1}, s_{i+1}, \alpha_{i+2}, \dots \alpha_r, s_r$ a partial trace. We define that an execution trace $\sigma$ satisfies a formula $\psi$, written as $\sigma \vDash_e \psi$, as follows:*

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

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

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

**Definition 7.** *Valuation of the Set of Event Variables. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, $\sigma = s_0, \alpha_1, s_1, \alpha_2, \dots \alpha_n, s_n$ a finite execution trace of T and $\mathcal{A}$ the set of event variables then we define the function $val_{\mathcal{A}}(\sigma)$ as follows:*

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

*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.*

In fact, we can represent an execution trace by an EOL formula. Suppose we want to represent the execution trace $\sigma$ = "Ta, Ca, Gf, Cc, Tc" by an EOL formula. We partition the set $\mathcal{A}$ of event variables in the set $Z$ containing all the event variables of the events that occur on $\sigma$ and the set $W$ containing all the event variables of the events that do not occur on $\sigma$. Consequently, $Z$ contains Ta, Ca, Gf, Cc, and Tc. The resulting EOL formula over $Z$ is $\psi = \text{Ta} \wedge \text{Ca} \wedge \text{Gf} \wedge \text{Cc} \wedge \text{Tc}$.

**Definition 8.** *Event Order Logic (EOL) Formula over Executions. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ \dots\ \alpha_n\ s_n$ an execution trace of T. The EOL over the execution $\sigma$ denoted by $\psi_\sigma$ 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 $\sigma$ and W contains all event variables of the events that do not occur on $\sigma$. $\psi_\sigma$ is the EOL formula containing all events in Z in the order they occur on $\sigma$ (e.g. $\psi_\sigma = a_{\alpha_1} \wedge a_{\alpha_2} \wedge \dots \wedge a_{\alpha_n}$).*

Now that we have established the formal basis to reason about the occurrence of events we have to formally define the notion of causality that we will use. A commonly adopted notion of causality is that of *counterfactual* reasoning and the related *alternative world* semantics of Lewis [5, 11]. 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. In our setting possible system execution traces represent the alternative worlds.

The *structural equation model (SEM)* by Halpern and Pearl [6] extends the counterfactual reasoning approach by Lewis. 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 order to do so the counterfactual test is extended by contingencies. Contingencies can be viewed as possible alternative worlds, where a variable value is changed. A variable X is causal if there exists a contingency, that is a variable valuation for other variables, that makes X counterfactual. In our precursory work [7], we extended the SEM by considering the order of the occurrences of events as possible causal factors. We now present an adaption of the SEM that can be used to decide whether a given EOL formula $\psi$ describes the causal process of the violation of some LTL formula $\varphi$ in a transition system. The causal process [6] comprises the causal events for the property violation and all events that mediate between the causal events and the property violation. Those events which are not root causes, are needed to propagate the cause through the system until the property violation is being triggered. If $\psi$ describes the causal process of a property violation we also say $\psi$ is causal for the property violation.

In a naive causality checking algorithm we perform the tests defined in Definition 9 for the induced EOL formula $\psi_\sigma$ of each $\sigma \in \Sigma_B$. The disjunction of all $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_n}$ that satisfy the conditions AC1-AC3 is the EOL formula describing all possible causes of the hazard.

**Definition 9.** *Cause for a Property Violation (Adapted SEM). Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and $\sigma$, $\sigma'$ and $\sigma''$ some execution traces of $T$. We partition the set of event variables $\mathcal{A}$ into sets $Z$ and $W$. An EOL formula $\psi$ consisting of the event variables in $Z$ is considered a cause for an effect represented by the violation of the LTL formula $\varphi$, if the following conditions are satisfied:*

- *AC1: There exists an execution $\sigma$, for which both $\sigma \vDash_e \psi$ and $\sigma \nvDash_l \varphi$ hold.*
- *AC2 (1): $\exists \sigma'$ s.t. $\sigma' \nvDash_e \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ and $\sigma' \vDash_l \varphi$. In words, there exists an execution $\sigma'$ where the order and occurrence of events is different from execution $\sigma$ and $\varphi$ is not violated on $\sigma'$.*
- *AC2 (2): $\forall \sigma''$ with $\sigma'' \vDash_e \psi \wedge (val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma''))$ it holds that $\sigma'' \nvDash_l \varphi$ for all subsets of $W$. In words, for all executions where the events in $Z$ have the value defined by $val_Z(\sigma)$ and the order defined by $\psi$, the value and order of an arbitrary subset of the events in $W$ have no effect on the violation of $\varphi$.*
- *AC3: The EOL formula $\psi$ is minimal: no subset of $\psi$ satisfies conditions AC1 and AC2.*

If we want, for instance, to show that $\psi = \text{Ta} \mathbin{\dot\wedge} \text{Ca} \mathbin{\dot\wedge} \text{Gf} \mathbin{\dot\wedge} \text{Cc} \mathbin{\dot\wedge} \text{Tc}$ is causal, we need to show that AC1, AC2(1), AC2(2) and AC3 are fulfilled for $\psi$.

- AC1 is fulfilled, since there exists an execution $\sigma$ = "Ta, Ca, Gf, Cc, Tc" for which $\sigma \vDash_e \psi$, and both the train and the car are in the crossing at the same time.
- AC2(1) is fulfilled since there exists an execution $\sigma'$ = "Ta, Ca, Gc, Tc" for which $\sigma' \nvDash_e \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \wedge val_W(\sigma) \neq val_W(\sigma'))$ holds and $\sigma'$ does not violate the property.
- Now we need to check the condition AC2(2). For the execution $\sigma''$ ="Ta, Ca, Gf, Cc, Cl, Tc" and the partition $Z, W \subseteq \mathcal{A}$, $\sigma'' \vDash_e \psi$ and $val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma'')$ hold. The property is not violated since the car leaves the crossing (Cl) before the train enters the crossing (Tc). As a consequence, AC2(2) is not fulfilled by $\psi$ because if Cl occurs between Cc and Tc, the property violation is prevented.

The example showed that the non-occurrence of events can be causal as well, and that this is not yet captured by the adapted SEM. The non-occurrence of an event is causal when ever AC1 and AC2(1) are fulfilled but AC2(2) fails for a EOL formula $\psi_\sigma$. If AC2(2) fails there is at least one event $\alpha$ on $\sigma''$ which did not occur on $\sigma$ and the occurrence of $\alpha$ prevents the property violation. Consequently, the non-occurrence of $\alpha$ on $\sigma$ is causal. We need to reflect the causal effect of the non-occurrence of $\alpha$ in $\psi_\sigma$. For the models that we analyze there are three possibilities for such a preventing event $\alpha$ to occur, namely,

1. at the beginning of the execution trace,
2. at the end of the execution trace, or
3. between two other events $\alpha_1$ and $\alpha_2$.

Furthermore, it is possible that the property violation is prevented by more than one event, hence we need to find the minimal set of events that are needed to prevent the property violation. This is achieved by finding the minimal true subset $Q \subset W$ of event variables that need to be changed in order to prevent the property violation.

**Definition 10.** *Non-Occurrence of Events. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and $\sigma$ and $\sigma''$ execution traces of $T$. We partition the set of event variables $\mathcal{A}$ into sets $Z$ and $W$. Let $\psi$ an EOL 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 execution $\sigma$ is causal for the violation of the LTL formula $\varphi$ if $\psi$ satisfies AC1 and AC2(1) but violates AC2(2), and if $Q$ is minimal, which means that there is no true subset of $Q$ for which $\sigma'' \models \psi \wedge val_Z(\sigma) = val_Z(\sigma'') \wedge val_Q(\sigma) \neq val_Q(\sigma'') \wedge val_{W \setminus Q}(\sigma) = val_{W \setminus Q}(\sigma'')$ and $\sigma'' \not\models \varphi$ holds.*

For each event variable $a_\alpha \in Q$ we determine the location of the event in $\psi''$ and prohibit the occurrence of $\alpha$ in the same location in $\psi$. We add $\neg a_\alpha \dot{\wedge}_]$ at the beginning of $\psi$ if the event occurred at the beginning of $\sigma''$ and $\dot{\wedge}_[ \neg a_\alpha$ at the end of $\psi$ if the event occurred at the end of $\sigma''$. If $\alpha$ occurred between the two events $\alpha_1$ and $\alpha_2$ we insert $\dot{\wedge}_< \neg a_\alpha \dot{\wedge}_>$ between the two event variables $a_{\alpha_1}$ and $a_{\alpha_2}$ in $\psi$. Additionally, each event variable in $Q$ is added to $Z$. In our example, $Cl$ is the only event that can prevent the property violation on $\sigma$ and occurs between the events $Cc$ and $Tc$. Consequently $\neg Cl$ is added to Z and $\psi$ and we get $\psi = \mathrm{Ta} \dot{\wedge} \mathrm{Ca} \dot{\wedge} \mathrm{Gf} \dot{\wedge} \mathrm{Cc} \dot{\wedge}_< \neg \mathrm{Cl} \dot{\wedge}_> \mathrm{Tc}$.

If a formula $\psi$ meets conditions AC1 through AC3, the occurrence of the events included in $\psi$ is causal for the violation of $\varphi$. However, condition AC2 does not imply that the order of the occurring events is causal. For instance, we do not know whether Ta occurring before Ca is causal in our example or not. If the order of the events is not causal, then there has to exists an execution for each ordering of the events that is possible in the system, and these executions all violate the property. Whether the order of events is causal is checked by the following Order Condition (OC1). Note that the outcome of OC1 has no effect on $\psi$ being causal, but merely indicates whether in addition the order of events in $\psi$ is causal.

**Definition 11.** *Order Condition (OC1). Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* $\sigma$, $\sigma'$ *execution traces of* $T$*. Let* $\psi$ *an EOL formula over* $Z$ *that holds for* $\sigma$ *and let* $\psi_{\wedge}$ *the EOL formula that is created by replacing all* $\curlywedge$*-operators in* $\psi$ *by* $\wedge$*-operators. The* $\curlywedge_{[}$*,* $\curlywedge_{]}$*, and* $\curlywedge_{<}\ \phi\ \curlywedge_{>}$ *are not replaced in* $\psi_{\wedge}$*.*

*OC1: The order of a subset of events* $Y \subseteq Z$ *represented by the EOL formula* $\chi$ *is not causal if the following holds:* $\sigma \vDash_e \chi \wedge \exists \sigma' \in \Sigma_B : \sigma' \nvDash_e \chi \wedge \sigma' \vDash_e \chi_{\wedge}$*.*

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 $\psi = \mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \curlywedge \mathrm{Cc})) \curlywedge_{<} \neg\mathrm{Cl} \curlywedge_{>} \mathrm{Tc})$.

## 3 On-The-Fly Causality Checking

### 3.1 Preliminaries

In order to compute causality relationships, it is necessary to compute good and bad execution traces. If depth-first search or breadth-first search is used for model checking, good and bad executions can easily be retrieved by the counterexample reporting capabilities of the model checker in use.

The key idea of the proposed algorithm is that the conditions AC1, AC2(1), AC2(2) and AC3 defined in Section 2 can be mapped to computing sub- and superset relationships between good and bad execution traces. In the following we also use the terms sub-execution and super-execution to refer to sub- or superset relationships between execution traces. We define a number of execution trace comparison operators as follows.

**Definition 12.** *Execution Trace Comparison Operators. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* $\sigma_1$ *and* $\sigma_2$ *execution traces of* $T$*.*

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

In the following let $\varphi$ a safety property specification given in LTL, $\sigma, \sigma', \sigma'', \sigma'''$ execution traces and $\psi_{\sigma}, \psi_{\sigma'}, \psi_{\sigma''}, \psi_{\sigma'''}$ the event order logic formulas representing these execution traces, respectively.

**Theorem 1.** *AC1 is fulfilled for all* $\psi_{\sigma}$ *where* $\sigma \in \Sigma_B$*.*

*Proof.* For each $\sigma \in \Sigma_B$ we can partition the set $\mathcal{A}$ of event variables into the sets $Z$ and $W$ such that $Z$ consists of the variables of the events that occur on $\sigma$ and $\psi_{\sigma}$ consists of the variables in $Z$. Consequently, $\sigma \vDash_e \psi_{\sigma}$ and $\sigma \nvDash_l \varphi$ because $\sigma$ is a bad execution. Therefore, AC1 is fulfilled for all $\psi_{\sigma}$ where $\sigma \in \Sigma_B$. □

**Theorem 2.** *AC2(1) holds for $\psi_\sigma$ if there is an execution $\sigma' \in \Sigma_G$ with $\sigma' \subset \sigma$.*

*Proof.* To show AC2(1) for a execution $\sigma$ we need to show that there exists an execution $\sigma'$ for which $\sigma' \nvDash_e \psi_\sigma \wedge (val_\sigma(Z) \neq val_{\sigma'}(Z) \vee val_\sigma(W) \neq val_{\sigma'}(W))$ and $\sigma' \vDash_l \varphi$ holds. For each $\sigma' \in \Sigma_G$ with $\sigma' \subset \sigma$ there is at least one event on $\sigma$ that does not occur on $\sigma'$. Because that missing event is part of $\psi_\sigma$ and $Z$ it follows $\sigma' \nvDash_e \psi_\sigma$ and $(val_\sigma(Z) \neq val_{\sigma'}(Z) \vee val_\sigma(W) \neq val_{\sigma'}(W))$ follows, since the value of the event variable representing the missing event assigned by $val_\sigma(Z)$ is *true* and the value assigned by $val_{\sigma'}(Z)$ is *false*. Therefore, we can show AC2(1) for $\psi_\sigma$ by finding an execution $\sigma' \in \Sigma_G$ for which $\sigma' \subset \sigma$ holds. □

**Theorem 3.** *AC2(2) holds for $\psi_\sigma$ if there is no execution $\sigma'' \in \Sigma_G$ with $\sigma \dot{\subset} \sigma''$.*

*Proof.* AC2(2) requires that $\forall \sigma''$ with $\sigma'' \vDash_e \psi_\sigma \wedge (val_\sigma(Z) = val_{\sigma''}(Z) \wedge val_\sigma(W) \neq val_{\sigma''}(W))$ it holds that $\sigma'' \nvDash_l \varphi$ for all subsets of $W$. Suppose there exists a $\sigma''$ for which $\sigma \dot{\subset} \sigma''$ holds. For a $\sigma''$ to satisfy the condition $\sigma'' \vDash_e \psi \wedge val_\sigma(Z) = val_{\sigma''}(Z)$ all events that occur on $\sigma$ have to occur in the same order on $\sigma''$, which is the case if $\sigma \dot{\subseteq} \sigma''$ holds. The set $W$ contains the event variables of the events that did not occur on $\sigma$ and $val_\sigma(W)$ assigns *false* to all event variables in $W$. For $val_{\sigma''}(W)$ to be different from $val_\sigma(W)$ there has to be at least one event variable that is set to *true* by $val_{\sigma''}(W)$. This is only the case if an event that does not occur on $\sigma$ occurs on $\sigma''$. Consequently, $\sigma''$ consists of all events that did occur on $\sigma$ and at least one event that did not occur on $\sigma$, which is true if $\sigma \dot{\subset} \sigma''$ holds. $\sigma'' \nvDash_l \varphi$ holds if $\sigma'' \in \Sigma_B$ and is false if $\sigma'' \in \Sigma_G$. Hence, AC2(2) holds for $\sigma$ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{\subset} \sigma''$ holds. □

**Theorem 4.** *If AC1 and AC2(1) hold for $\psi_\sigma$ and $\psi_\sigma$ is modified according to Def. 10 in order to fulfill AC2(2), then AC1 and AC2(1) hold for the modified $\psi_\sigma$.*

*Proof.* The modification defined in Def. 10 prohibits the occurrence of events that did not occur on $\sigma$ but occur on $\sigma''$ by adding their corresponding negated event variables to $\psi_\sigma$. Since the prohibited events did not occur on $\sigma$, the modified $\psi_\sigma$ holds for $\sigma$ and AC1 holds.

AC2(1) holds for the modified $\psi_\sigma$ because for AC2(1) to hold in the first place there has to be an execution $\sigma' \in \Sigma_G$ with $\sigma' \subset \sigma$. For the modification of $\psi_\sigma$ to be necessary an execution $\sigma'' \in \Sigma_G$ with $\sigma \dot{\subset} \sigma''$ has to exist. If $\sigma \dot{\subset} \sigma''$ holds, $\sigma \subset \sigma''$ holds and $\sigma' \subset \sigma''$ holds as well. Consequently, AC2(1) holds for the modified $\psi_\sigma$. □

**Theorem 5.** *AC(3) holds for $\psi_\sigma$ if there does not exists an execution $\sigma''' \in \Sigma_B$ for which $\sigma''' \subset \sigma$ holds.*

*Proof.* In AC(3) we have to show that no subset of the event order logic formula $\psi$ satisfies AC1, AC2(1) and AC2(2). Suppose there exists a $\sigma''' \in \Sigma_B$ 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 $\sigma'''$ and $\psi_{\sigma'''}$ consists of the variables in $Z_{\sigma'''}$. For $\sigma$ we partition $\mathcal{A}$ in $Z_\sigma$ and $W_\sigma$ such that $Z_\sigma$ consists of the variables of the events that occur on $\sigma$ and $\psi_\sigma$ consists of the variables in $Z_\sigma$. 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. If we can not find a $\sigma'''$with $\sigma''' \subset \sigma$, then no subset of $\psi_\sigma$ satisfies AC1, AC2(1) and AC2(2), and consequently AC3 holds. □

We use these theorems in order to devise an algorithm and a corresponding data-structure called subset graph for on-the-fly causality checking. The pseudo-code for the proposed algorithms can be found in [12].

### 3.2 Subset Graph Data-Structure

In order to store the execution traces we have devised a data-structure called subset graph. This data-structure enables us to make causality decisions on-the-fly which means that we can decide whether an execution trace is causal as soon as we add it to the subset graph. The subset graph is structured into levels where each level corresponds to the length of the execution traces on that level. Each node represents exactly one execution trace. Figure 1 shows a part of the subset graph for the railroad crossing example. The execution traces on adjoining levels are connected by edges indicating subset relationships between the respective execution traces. In order to improve readability the edges between executions on the same level are not displayed in the figure. The nodes representing the

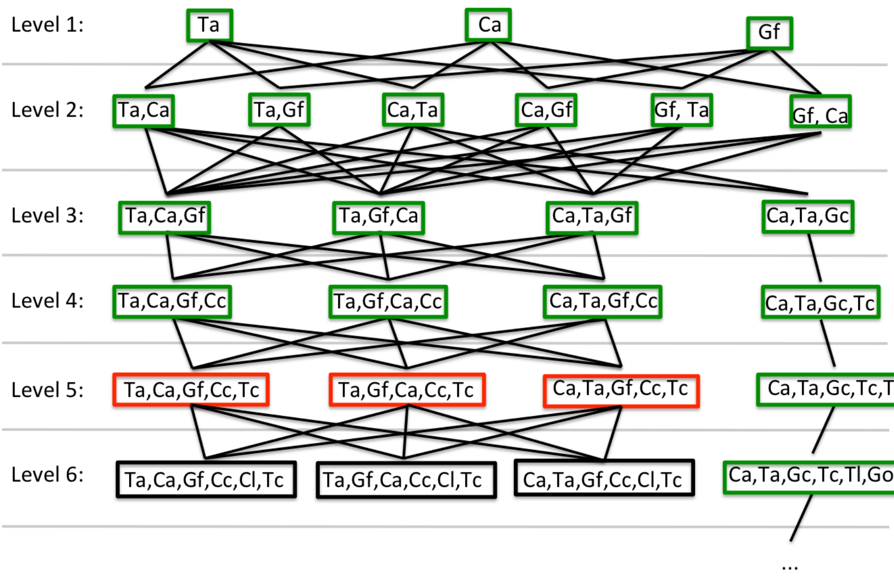

**Fig. 1.** Subset-graph of the railroad crossing example.

execution traces are colored in green, red, black or orange in order to indicate their potential causality relation according to the following rules:

– Green: a node is colored *green* if it represents a good execution trace and all nodes on the level below that are connected with it are also colored green. An example of such a trace is “Ca,Ta,Gc,Tc,Tl” in the railroad crossing example. Green traces can not be causal because they are good traces. The green traces can be prefixes of either bad or good execution traces.
– Red: a node is colored *red* if it represents a bad execution trace and all nodes on the level below that are connected with it are colored green. Red nodes correspond to the shortest bad traces found at any point of the state-space exploration. They are considered to be causal. As an example consider the trace “Ta,Ca,Gf,Cc,Tc” in the railroad crossing example.
– Black: a node is colored *black* if it represents a good execution trace, but at least one node on the level below that it is connected with is colored red. Black traces cannot be causal themselves, since they are good traces, but since a sub-trace of them with one less element is a minimal bad trace, the transition in the subset graph from red to black identifies an event that turns a bad execution into a good one. We hence take advantage of black traces when checking condition AC2(2). As an example for a black node consider the the trace “Ta,Ca,Gf,Cc,Cl,Tc” of the railroad crossing example, which is connected with the red execution “Ta,Ca,Gf,Cc,Tc” on the level below, the introduced “Cl” event turns the bad execution into a good one.
– Orange: A node is colored *orange* if it represents a bad execution trace and at least one node on the level below that is connected to the orange node is colored red. If a trace is colored orange, there exists a shorter red trace on a level below and hence a orange trace does not fulfill the minimality constraint AC3 for being causal. An example for an orange colored trace is the trace “Ca,Ta,Gc,Tc,Tl,Go,Ta,Gf,Cc,Tc” which, due to space restrictions, is not depicted in Figure 1. The trace “Ca,Ta,Gf,Cc,Tc” is a shorter red trace and a subset of the trace “Ca,Ta,Gc,Tc,Tl,Go,Ta,Gf,Cc,Tc”, hence the trace does not fulfill the minimality constraint.

## 3.3 Causality Checking

The causality checking that we propose is embedded into both of the standard state-space exploration algorithms used in explicit state model checking, namely depth-first and breadth-first search. Whenever a bad or a good execution is found by the search algorithm it is added to the subset graph. After adding a trace the algorithm first checks whether there are executions at the same level that consist of the same events but in a different order.

– If we find such an execution, then all subset relationships of the execution already in the subset graph hold also for the newly added execution. For instance in our example all subset relationships that hold for the execution “Ta,Ca,Gf,Cc,Tc” also hold for the execution “Ta,Gf,Ca,Cc,Tc”.
– If we don’t find such a trace on the same level, we have to check the subset relationships with the execution traces on the level below (level-1) and, if depth-first search is used, on the level above (level+1) as well. It is not

necessary to check the subset relationships on the level above (level+1) if breadth-first search is used, because breadth-first search adds the traces by increasing length, hence there are not yet any traces on the level above.

Once all subset relationships are established, the nodes representing the executions are colored according to the above described coloring rules. If a trace is colored red, we additionally need to check whether we have already found a shorter red trace which is a sub-set of the new red-trace. If such a shorter red trace is found, the current trace is colored orange. In our example the execution traces *Ta, Ca, Gf, Cc, Tc* and *Ta, Gf, Ca, Cc, Tc* and *Ca, Ta, Gf, Cc, Tc* are colored red and hence considered to be causal.

The following theorems show that for an execution $\sigma$ that is colored red, $\psi_\sigma$ is a candidate for being causal and fulfills AC1, AC2(1) and AC3.

**Theorem 6.** *AC1 is fulfilled for $\psi_\sigma$ of each execution trace $\sigma$ that is colored red.*

*Proof.* By definition an execution trace is only colored red if it is a bad trace and according to Theorem 1 AC1 is fulfilled for all $\sigma \in \Sigma_B$. □

**Theorem 7.** *AC2(1) is fulfilled for $\psi_\sigma$ of each execution trace $\sigma$ that is colored red.*

*Proof.* According to Theorem 2 we can show AC2(1) by finding an execution $\sigma' \in \Sigma_G$ for which $\sigma' \subset \sigma$ holds. For an execution $\sigma$ to be colored red, all sub execution traces on the level below have to be colored green. Consequently, for each execution $\sigma'$ for which $\sigma' \subset \sigma$ holds also $\sigma' \in \Sigma_G$ holds because it is colored green and hence needs to be a good trace. Therefore, AC2(1) is fulfilled according to Theorem 2. □

**Theorem 8.** *If breadth-first search is used, AC3 is fulfilled for $\psi_\sigma$ of each execution trace $\sigma$ that is colored red. If depth-first search is used, AC3 is fulfilled for $\psi_\sigma$ of each execution trace $\sigma$ that is colored red as soon as the state-space exploration has terminated.*

*Proof.* According to Theorem 5, $\psi$ fulfills AC3 if there does not exists a trace $\sigma''' \in \Sigma_B$ for which $\sigma''' \subset \sigma$ holds. This is due to the fact that by definition an execution trace is only colored red if all its subsets are colored green, which means there is no bad sub-execution $\sigma'''$ of $\sigma$. If breadth-first search is used the shortest paths are added first, hence all sub-executions are known at the time where $\sigma$ is inserted and colored. Consequently, if breadth-first search is used, AC3 is fulfilled for $\psi_\sigma$ of each execution trace $\sigma$ that is colored red. If depth-first search is used it is possible that new sub-executions are found as long as the state-space exploration is not complete. As a result, AC3 is fulfilled for $\psi_\sigma$ of each execution trace $\sigma$ that is colored red as soon as the state-space exploration with depth-first search has terminated. □

Once the state space search is completed we have to perform the tests for AC2(2) and OC1 for all red execution traces.

According to Theorem 3, AC2(2) holds for $\psi_\sigma$ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{\subset} \sigma''$ holds. If such a $\sigma''$ exists, it is a black superset of $\sigma$ because $\sigma \subset \sigma''$ holds for each black superset of $\sigma$. $\sigma''$ is only colored black if it is a good trace. Consequently, we need to check for each black superset $\sigma''$ of $\sigma$ whether $\sigma \dot{\subset} \sigma''$ holds. If there is no $\sigma''$ for which $\sigma \dot{\subset} \sigma''$ holds, then $\psi_\sigma$ fulfills AC2(2). If $\sigma \dot{\subset} \sigma''$ holds for a black superset, then we need to modify $\psi_\sigma$ as specified by Definition 10. Hence, we have shown that AC1, AC2(1), AC2(2) and AC3 are fulfilled for $\psi_\sigma$ of each red execution $\sigma$ and, consequently, that $\psi_\sigma$ is causal for the property violation.

Notice that the AC2(2) test is needed in order to detect whether the non-occurrence of an event is causal. It is necessary to store all traces that are colored black only to test AC2(2). We have added a runtime switch in the implementation of the causality checking method that allows the user to turn the AC2(2) test off in order to save memory at the expense of not being able to take the possible causality of the non-occurrence of an event into account. If the AC2(2) test is fulfilled by $\psi_\sigma$, then the OC1 test is performed. Due to the structure of the subset graph, it is sufficient to check for each red execution trace whether there exists a red execution trace on the same level for which the unordered $\subseteq$ relationship holds. For all those execution traces, we check for each pair of events whether they appear on all execution traces in the same order or not. If a pair of events does not occur in the same order, then the order of this pair is marked as having no influence on causality.

**Algorithm 1** Pseudo code of the Sub-Set Graph data structure

```
class SubSetGraph {

HashMap <Integer, Trace> traces;
List of List of Integers levelToIDs;
List of Integers = redTraces;
boolean searchNotCompleted = true;

  addTrace(Trace t) {
    // ... see extra figure
  }
}
```

## 3.4 Causality Checking with Breadth-First Search (BFS)

The pseudo-code of the adapted breadth-first search algorithm is given in the Algorithm 6 figure. When using breadth-first search, the execution trace leading from the initial state to a property violating state can be generated by iterating

**Algorithm 2** Algorithm sketch of the addTrace() method of the sub-set graph (part 1)

```
addTrace(Trace t)
{
   t.color = green;
   if(t.isBad()) {
      add(redTraces, t);
      t.color = red;
   }

  // add trace to its level
  add(levelToIDs, length(t), getID(t));

    boolean subSetSearchNeeded = true;
   // same level check for OC 1*/
   int level = length(t);
   for each t' in getTracesByLevel(level){
          Trace tempTrace;

          if(isSubSet( t', t)) {
               addSameLevelSuperSet(t', t)
               addSameLevelSubSet(t, t')
              //If t1 has sem length and is subset,
              //all subsets of t1 are subsets of t
              //subset search is not needed
              subSetSearchNeeded = false;
              t.SubSets = t'.SubSets;
              t.SuperSets = t'.SuperSets;
          }
   }

  /* Find all direct subsets */
  level = level - 1;

  while(subSetSearchNeeded && level >= 0) {
      boolean subsetFound = false;
      for each t' in getTracesByLevel(level) {
          Trace tempTrace;

          if(isSubSet( t', t))
          {
               addBlackSuperSet(t', t)
               addBlackSubSet(t, t')

            if(t'.color != green && t.isBad()){
                 t.color = orange;
                remove(redTraces, t);
            }else{
                 t.color = black;
            }
            subsetFound = true;
          }
        }
        if(!subsetfound) {
           //no subset found at current level
           level--;
        } else {
           level = length(t)-level;
       }
  }
```

**Algorithm 3** Algorithm sketch of the addTrace() method of the sub-set graph (part 2)

```
  if(t.color == red)
  {
    //check for all red traces whether
    //there is discontinued red trace
    //which is a subset of t
    FOR ALL t_1 in redTraces
    {
      if(t_1.getID() != t.getID()
       && t.getLength() > t_1.getLength()
       && t.isSubset(t_1))
      {
        t.color = orange;
        remove(redTraces, t);
      }
    }
  }
}
```

backwards through the predecessor links until an initial state is reached. Whenever a bad or a good execution is found, it is added to the subset graph. If BFS encounters a state that is already in the state-space and hence all successors of this state have already been explored, the successors are not explored for a second time. Since BFS explores the state-space following an exploration order that leads to a monotonically increasing length of the execution traces, this new execution trace reaching the state either has the same length as the already known execution trace containing the same state, or the new execution is longer than the already known execution trace. If the new execution trace has the same length, the events on the trace have an order that is different from the one in the already known execution trace. Hence the new execution trace needs to be added to the subset graph since a later OC1 test needs to be performed on it.

## 3.5 Causality Checking with Depth-First Search (DFS)

We adapted the depth-first search algorithm to add an execution trace to the subset graph data structure whenever either a bad state is reached or a good execution trace has been found. If depth-first search is used it is sufficient to print the search stack in order to retrieve the execution trace. Similarly to BFS, if DFS encounters a duplicate, which is a state that is already in the state-space, and hence all successors of the duplicate have already been explored, the successors are not explored a second time. It is possible that this new trace to the duplicate is shorter or has a different event order than the already known execution traces that contain the duplicate. Hence we store this new execution

**Algorithm 4** Algorithm sketch of the checkAC22() method.

```
function checkAC22()
{
  FOR ALL Trace t in redTraces
  {
    FOR ALL Trace t_1 in
        getTracesOnLevel(length(t)+1)
    {
      if(isSubset(t, t_1))
      {
        List events_t = t.getEvents();
        List events_t_1 = t_1.getEvents();
        u = 0;
        FOR x = 0 to length(p)
        {
          if(events_t.get(u) = events_t_1.get(x)))
          {//events are same move to next event
            u++;
          }
          else
          {//events are different negate event x
            t_1.negateEvent(x);
          }
        }
        //Replace t with t_1 containing
        //negated events
        redTraces.replace(t, t_1);
      }
    }
  }
}
```

**Algorithm 5** Algorithm sketch of the checkOC1() method.

```
function checkOC1()
{
  FOR ALL Trace t in redTraces
  {
    //mark all pairs as ordered
    FOR i = 0 to length(t)
    {
      FOR i = j to length(t)
      {
        t.MarkOrdered(i,j,true);
      }
    }

    FOR ALL t_1 in getTracesOnLevel(length(t))
    {
      if (t.getID() != t_1.getID())
      {
        if(isEqual(t, t_1)
        {
          events_t = t.getEvents();
          events_t_1 = t_1.getEvents();

          if(isOrderedEqual(t, t_1))
          {
            redTraces.remove(t);
          }
          else
          {
            FOR i = 0 to size(events_t)
            {
              FOR i = j to size(events_t)
              {
                if(!(
                 events_t_1.indexOf(events_t.get(i))
                 <=
                 events_t_1.indexOf(events_t.get(j))
                ))
                {//pair i,j is not ordered
                  t.MarkOrdered(i,j,false);
                }
              }
            }
          }
        }
      }
    }
  }
}
```

**Algorithm 6** Algorithm sketch of the adapted breadth first search algorithm.

```
Queue D = {}, State-space V = {}, SubSetGraph G = {}

function main()
{
  add(V, init_state)
  add(D, init_state)
  bfs()

  checkAC22();
  checkOC1();
}

function bfs()
{
  //Returns top element of the queue and deletes it.
  s = getHead(D);

  if error(s)
  { //bad trace found
     trace = buildTrace(s)
     G.addTrace(trace,1)
  }
  else
  {  //"so far good" trace found
     trace = buildTrace(s)
     G.addTrace(trace,0)
  }

  if hasNoSuccessors(s) & NOT error(s)
  { //good trace found
     trace = buildTrace(s)
     G.addTrace(trace,0)
  }

  while( s has successor t & G.searchNotCompleted )
  {
    if in(V, t) == false
    { // this is for generating the traces
      // (backwards linking)
      setPrevious(t,s)
      add(V,t)
      add(D,t)
      bfs()
    }
    else
    { // found a new path to t
        trace = buildTraceFromStack(Stack)
        G.addToMatchList(trace,0)
       /*Found new path to already known state,
       add trace to match list*/
    }
}
```

trace on a match list in the subset graph and generate all execution traces starting from the duplicate state with the new trace as a prefix.

### 3.6 Complexity

[13] 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 is NP-complete. Results in [14] show that causality can be computed in polynomial time if the causal graph over the events forms a directed causal tree. A directed causal tree consists of directed paths, where the nodes represent events, and the edges represent the causality relationships and the root node represents the hazard or effect. Each bad execution 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.

## 4 Case Studies

### 4.1 Experiment Setup

In order to evaluate the proposed approach, we have implemented our causality checking algorithms within the SpinJa toolset [15], a Java re-implementation of the explicit state model checker Spin [8]. Our SpinCause tool[1] computes the causality relationships for a Promela model and a given LTL property. In order to compute all interleavings and all executions partial-order reduction was disabled during the state-space exploration. The Promela models used for the case studies have been created manually, in practical usage scenarios the Promela models can also be automatically synthesized from higher-level design models, as for instance by the QuantUM tool [16]. The following experiments were performed on a PC with an Intel Xeon Processor (3.60 Ghz) and 144 GBs of RAM.

### 4.2 Railway Crossing

The Promela model of the railway crossing that we constructed as a running example for the purpose of this paper comprises 133 states and 237 transitions. A total of 47 bad execution traces are found. The causality checking algorithm identified two event order logic formulas describing the causal factors for a train and a car being on the crossing at the same time.

– First, if the gate fails 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 $(\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\dot\wedge} \mathrm{Cc})) \mathbin{\dot\wedge}_{<} \neg \mathrm{Cl} \mathbin{\dot\wedge}_{>} \mathrm{Tc}))$.

[1] http://se.uni-konstanz.de/research1/tools/spincause

**Algorithm 7** Algorithm sketch of the extended depth first search algorithm.

```
SubSetGraph G = {}
State-space V = {}
Stack S = {}

function main(s)
{
  dfs(init_state)
  checkAC22()
  checkOC1()
}

function dfs(s)
{
  if error(s)
  {
    report error
    trace = buildTrace(s)
    G.addTrace(trace,1)
    /*bad execution trace found,
     add to causality computation*/
  }
  else
  {
          trace = buildTraceFromStack(Stack)
          G.addTrace(trace,0)
         /*"so far good" execution trace found,
          add to causality computation*/
  }
  add(V,s)
  add(S,s)

  if hasNoSuccessors(s) & NOT error(s)
  {
    trace = buildTraceFromStack(Stack)
    G.addTrace(trace,0)
    /*good execution trace found,
     add to causality computation*/
  }

  for each successor t of s
  {
    if  in(V, t) == false
    {
      dfs(t)
    }
    else
    {
          trace = buildTraceFromStack(Stack)
          G.addToMatchList(trace,0)
         /*Found new path to already known state,
          add trace to match list*/
    }
  }
 delete s from Stack
}
```

– Second, 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 $((\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\dot\wedge} \mathrm{Cc})) \mathbin{\dot\wedge}_{<} \neg \mathrm{Cl} \mathbin{\dot\wedge}_{>} (\mathrm{Gc} \wedge \mathrm{Tc}))$.

## 4.3 Airbag Control Unit

The industrial size model of an airbag system that we use in this case study is taken from [17]. The architecture of this system was provided by our industrial partner TRW Automotive GmbH. The architecture of this system consists of two acceleration sensors, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag. Although airbags save lifes in crash situations, they may cause fatal accidents if they are inadvertently deployed. This is because the driver may lose control of the car when this deployment occurs. 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 the hazard corresponding to an inadvertent ignition of the airbag. The Promela model of the airbag system consists of 155,464 states and 697,081 transitions.

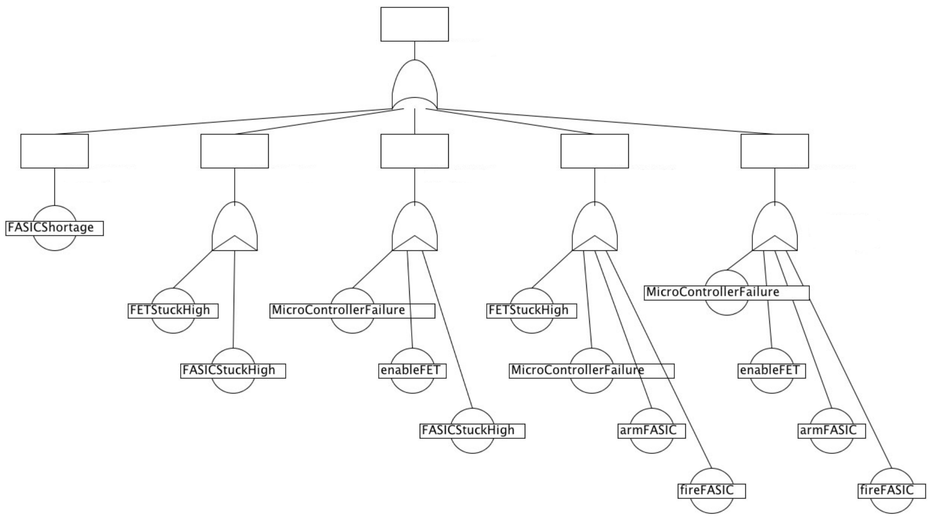

**Fig. 2.** Fault tree of the airbag system

Figure 2 shows the fault tree generated by the SpinCause tool. All execution traces that are colored red are part of the fault tree representation. The fault trees generated by our approach all have a normal form, that is they start with an *intermediate* gate representing the top level event, that is connected to an *OR* gate. The execution traces that are colored red are represented by Priority-AND (PAND) gates if the order of some events is causal and by AND gates if the order is not causal. The events of the execution traces are connected to the corresponding AND or PAND gates, respectively. Since fault trees are not sufficiently expressive to completely represent an event oder logic formula, we display for each PAND gate the event order logic formula constraining the order of the events connected to the PAND-gate (omitted in Figure 2 for better readability).

While there are a total of 20,300 bad execution traces, the fault tree comprises only 5 paths. Obviously, a manual analysis of this large number of traces in order to determine causal factors would be impossible. It is easy to see in the fault tree which basic events cause an inadvertent deployment of the airbag. For instance, there is only one single fault that can lead to an inadvertent deployment, namely *FASICShortage*, which is represented by the event order logic formula FASICShortage. It is also obvious that the combination of the basic events *FETStuckHigh* and *FASICStuckHigh* only leads to an inadvertent deployment of the airbag if the basic event *FETStuckHigh* occurs prior to the basic event *FASICStuckHigh*, which is represented by the event order logic formula FETStuckHigh $\dot\wedge$ FASICStuckHigh. The basic event *MicroControllerFailure* can lead to an inadvertent deployment if it is followed by the following sequence of basic events: *enableFET*, *armFASIC*, and *fireFASIC*. This sequence is represented by the event order logic formula MicroControllerFailure $\dot\wedge$ enableFET $\dot\wedge$ armFASIC $\dot\wedge$ fireFASIC. If the basic event *FETStuckHigh* occurs prior to the *MicroControllerFailure* the sequence in which *armFASIC* and *fireFASIC* occur after the *MicroControllerFailure* event suffices to lead to the top level event. This sequence is represented by the event order logic formula FETStuckHigh $\dot\wedge$ MicroControllerFailure $\dot\wedge$ armFASIC $\dot\wedge$ fireFASIC. If the basic event *FASICStuckHigh* occurs after *MicroControllerFailure* and *enableFET* this also leads to a sequence leading to an inadvertent deployment. It is represented by the event order logic formula MicroControllerFailure $\dot\wedge$ enableFET $\dot\wedge$ 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. If the order of the events is important for the events causing the effect, this can be seen in the fault tree by the *PAND* gate and the corresponding EOL formula. In the counterexamples computed by SpinJa one would have to manually compare the order of the events in all execution traces.

## 4.4 Discussion

Table 1 shows the memory and run time consumption of the on-the-fly causality checking approach presented in this paper for both case studies and the memory and run time consumption of the in off-line approach presented in [7], where all execution traces are stored on disk during model checking (Run. MC., Mem. MC) and the causality checking is performed as a post-processing step (Run. Caus., Mem. Caus.), for the airbag case study. The following trends can be identified:

– If no causality checking is done, DFS and BFS have approximately the same runtime and memory consumption. The causality checking adds a run-time

|  | On-the-fly Approach |  |  |  |  |  | Off-line Approach |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  | Run time (sec.) |  |  | Memory (MB) |  |  | Run time |  | Memory (MB) |  |
|  | MC | CC 1 | CC2 | MC | CC1 | CC 2 | MC | Caus. | MC | Caus. |
| Airbag |  |  |  |  |  |  |  |  |  |  |
| DFS | 0.98 | 338.17 | 597.57 | 25.08 | 15,711.20 | 27,687.50 | 871.14 | 945.68 | 1,478.34 | 28,563.47 |
| BFS | 0.96 | 148.52 | 195.05 | 25.74 | 1,597.54 | 3,523.04 | 486.01 | 512.3 | 1,331.29 | 13,860.10 |
| Railway |  |  |  |  |  |  |  |  |  |  |
| DFS | 0.01 | 0.29 | 0.31 | 16.40 | 20.38 | 21.68 | - | - | - | - |
| BFS | 0.01 | 0.12 | 0.13 | 16.24 | 16.70 | 17.45 | - | - | - | - |

**Table 1.** This table shows the experiment results with the on-the-fly approach for the railway crossing and airbag case studies. Run. MC and Mem. MC show the runtime and memory consumption for model checking only. Run. CC1 and Mem. CC1 show the runtime and memory needed to perform model checking and causality checking with the AC2(2) test disabled and Run. CC2 and Mem. CC2 with the AC2(2) test enabled. Additionally, the experiment results for off-line causality checking of the airbag case study are given.

and memory penalty, but the experiments show that causality checking is applicable to industrial size Promela models. In addition causality checking provides valuable insight as to why the hazard occurred, which is very tedious or even impossible to determine if standard model checking and manual counterexample analysis is used.

– When performing causality checking, BFS outperforms DFS in terms of both runtime and memory consumption. BFS outperforms DFS because if BFS is used, we can safely rely on the assumption that when a bad trace is found all shorter bad traces already have been found. This assumption assures that the minimality condition holds for each bad trace which was found using BFS and colored red by the causality checking algorithm. If DFS is used, no assumptions on the length of the bad trace can be made. The main reason why the assumption on the bad trace length is important and has such a high impact on the memory consumption when using DFS compared to BFS is that all good traces which are supersets of a red trace have to be taken into account for the AC2(2) test. When BFS is used only the traces which are supersets of red traces need to be stored, whereas when DFS is used all good traces need to be stored. Because the good traces are needed in case a shorter red trace is found later in the search for which we need the good super-traces for the AC2(2) test.
– The on-the-fly approach proposed in this paper outperforms the off-line approach both in terms of runtime and memory consumption. The main reason for this observation is that when using the on-the-fly approach only the execution traces needed for causality checking, namely the red and black execution traces, need to be stored, whereas all execution traces have to be stored for the off-line approach.

## 5 Related Work

The application of counterfactual reasoning to software debugging has been proposed by Zeller in [18]. However, [18] does not support complex logical relationships as causes and is mainly applicable to sequential software programs, whereas our approach is also applicable to concurrent software and hardware systems.

Work documented in [19] uses the Halpern and Pearl approach to explain counterexamples in CTL model checking by determining causality. However, this approach considers only single counterexamples. 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. Consider the railway crossing example in which the CTL formula consists of the two boolean variables train_on_crossing and car_on_crossing. Obviously, both variables changing to true is causal for a crash. Consequently the approach from [19] will indicate the variable value-change of train_on_crossing and car_on_crossing from false to true as being causal. But this obvious answer does not give any insight on why the train and the car are on the crossing at the same time.

In [20] a formal framework for reasoning about contract violations is presented. In order to derive causality the notion of precedence established by Lamport clocks [21] is used. While this captures a partial order of the observed contract violations it is not clear to what extent this order information also expresses causality. Work described in [22] establishes causality based on 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. For all the above mentioned approaches it is necessary to compute the counterexamples prior to the causality analysis whereas our approach works on-the-fly. To the best of our knowledge we are not aware of any other causality checking algorithm that can be integrated with explicit state-space exploration algorithms and which works on-the-fly.

As an alternative to the event order logic that we defined we also investigated the usage of the interval logics [23] and [24]. We felt that in light of the relatively simple ordering constraints that we need to describe those logics are overly expressive, and we hence decided to define our own tailored, relatively simple event order logic.

## 6 Conclusions

We have discussed how causality relationships can be established in system executions and have shown how the causality checks can be mapped to finding sub- and super-sets of execution traces. Furthermore we have proposed an approach for causality computation that works on-the-fly and can be integrated into explicit state-space model checking algorithms. We have evaluated our approach on two case studies, one of which is of industrial size. The experimental evaluation indicates that breadth-first search outperforms depth-first search in terms of memory and runtime, and that the on-line approach presented here outperforms the precurosy off-line approach. Furthermore, we have shown that causality checking is applicable to industrial size Promela models.

In future work we plan to give a soundness and completeness argument for causality checking and embed causality checking into a symbolic reasoning environment in order to avoid the explicit storing of traces. In addition we plan to combine our work on causality checking for probabilistic models with the approach presented here.

*Acknowledgment.* We wish to thank Stefan Heindorf for a careful review of an earlier version of this work.

## References

1. IEEE, "Ieee std 1028-2008 standard for software reviews," 2008.
2. *Fault Tree Handbook*, U.S. Nuclear Regulatory Commission, 1981, nUREG-0492.
3. International Electrotechnical Commission, "Analysis Techniques for System Reliability - Procedure for Failure Mode and Effects analysis (FMEA), IEC 60812," 1991.
4. E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking (3rd ed.).* The MIT Press, 2001.
5. D. Lewis, *Counterfactuals.* Wiley-Blackwell, 2001.
6. J. Halpern and J. Pearl, "Causes and explanations: A structural-model approach. Part I: Causes," *The British Journal for the Philosophy of Science*, 2005.
7. 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*, ser. LNCS. Springer, 2011.
8. G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual.* Addision–Wesley, 2003.
9. C. Baier and J.-P. Katoen, *Principles of Model Checking.* The MIT Press, 2008.
10. Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., 1992. [Online]. Available: http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=128869
11. J. Collins, Ed., *Causation and Counterfactuals.* MIT Press, 2004.
12. F. Leitner-Fischer and S. Leue, "Causality checking for complex system models," Chair for Software Engineering, University of Konstanz, Technical Report soft-12-02, 2012, available from http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-12-02.pdf.
13. T. Eiter and T. Lukasiewicz, "Complexity results for structure-based causality," *Artificial Intelligence*, 2002.
14. ——, "Causes and explanations in the structural-model approach: Tractable cases," *Artificial Intelligence*, vol. 170, no. 6-7, pp. 542–580, 2006.
15. M. de Jonge and T. Ruys, "The spinja model checker," in *Model Checking Software*, ser. Lecture Notes in Computer Science, vol. 6349. Springer, 2010, pp. 124–128.
16. F. Leitner-Fischer and S. Leue, "QuantUM: Quantitative safety analysis of UML models," in *Proceedings Ninth Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, ser. EPTCS, vol. 57, 2011, pp. 16–30. [Online]. Available: http://www.inf.uni-konstanz.de/soft/research/publications/pdf/qapl2011.pdf

17. 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.
18. A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging.* Elsevier, 2009.
19. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," in *Proceedings of CAV 2009*, ser. LNCS. Springer, 2009.
20. 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.
21. 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
22. A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 3, 2006.
23. R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt, "An interval logic for higher-level temporal reasoning," in *Proceedings of the second annual ACM symposium on Principles of distributed computing.* ACM, 1983, pp. 173–186.
24. L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. Ramakrishna, "A graphical interval logic for specifying concurrent systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 3, no. 2, pp. 131–165, 1994.