An Algorithm to Compute a Strict Partial Ordering of Actions in Action Traces

Martin Kölbl and Stefan Leue

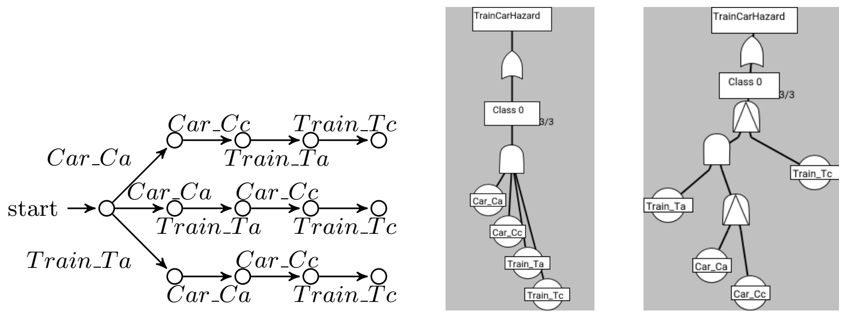
University of Konstanz, Germany

Abstract. Causality Checking [LL13] computes a causal explanation in the form of minimal action traces that lead to the violations of a reachability property. Causality Checking is implemented in the tool QuantUM [LFL11] that currently only depicts in a fault tree the causal actions in the action traces that lead to a property violation, but not the possible order of these actions. We present an analysis to compute the strict partial order of actions in action traces and succinctly depict these orders by a fault tree. We implemented the analysis in the tool QuantUM. We assess the performance of our algorithm by applying it to several models of different size. The results show that the analysis can compute the action order for thousands of action traces.

1 Motivation

Model-driven development is an efficient way to deal with the complexity of modern systems. A model is a high-level abstraction of a system and can support the development of a correct system. Before the implementation of a system, a model checker can verify a model of the system to ensure that the system behaves according to its specification. For the verification, the specification of a model is given as a property. An initial design typically has shortcomings and violates the property. When a model checker finds a violation of a property, it returns a counterexample in the form of an execution that leads to the violation. An execution contains an ordered sequence of actions that we call an action trace. We proposed Causality Checking in [LL13] that analyzes the counterexamples of a model based on the counterfactual argument [Lew01] and results in a set of action traces that are considered to be causal, according to the counterfactual actual cause definition given in [LL13]. When a system execution contains one of the action traces the property will be violated. A system execution that contains none of the action traces will not violate the property.

We implemented Causality Checking in the tool *QuantUM*. The input of QuantUM is a reachability property and a model in SysML [Obj17]. QuantUM converts the SysML model into the model checking language `Spin` [Hol04] and executes Causality Checking based on a systematic state space exploration to find every causal action trace in the model. Afterwards, QuantUM pools the causal action traces with the same set of actions to a causality class. The disjunction of the different causality classes constitutes the cause of the property



(a) Railroad Crossing (RC) Traces (b) RC Fault Tree (c) RC Ordered Fault Tree

Fig. 1. Railroad example

violation and will be displayed as a fault tree [KL19]. Currently, QuantUM depicts the set of actions in a causality class by a fault tree without indicating the order of the contained actions. In previous work, amongst others we applied Causality Checking to the architecture of a self-driving car [KL19], and showed that Causality Checking supports the safety assessment in the development of a safety-critical system.

We now illustrate the analysis performed during causality checking by applying it to the model of a railroad crossing, which we will refer to as a running example throughout the paper. In this model, a train approaches a crossing (Train_Ta), then enters the crossing (Train_Tc) and leaves the crossing (Train_Tl). A car also approaches at the crossing (Car_Ca), then enters the crossing (Car_Cc) and leaves the crossing (Car_Cl). The crossing is unguarded and has no gate. The car will not enter the crossing when the train is already in the crossing. A hazard in this system occurs when a state can be entered in which both the train and the car are in the crossing at the same time, which has the potential to lead to a fatal accident. We, therefore, state the property that such a state can not be reached.

Causality Checking computes causes for the violation of such a reachability property in case it is violated in the model. The result of this cause computation is a number of what is referred to as *causality classes*. The action traces in one causality class are all formed over the same set of actions, and only vary in the order in which these actions occur. For the railroad crossing example, Causality Checking computes just one causality class over the set {Train_Ta, Train_Tc, Car_Ca, Car_Cc} of actions. It contains 3 action traces, depicted in Figure 1(a). Notice that for systems of realistic size, a causality class may contain a much higher number of action traces. For reasons of convenience and since QuantUM is primarily used in the area of safety-critical system analysis, causality classes are depicted as fault trees. The fault tree in Figure 1(b) depicts the causality class computed for the Railroad Crossing example. The top-level event *Train-CarHazard* is valid when one of the causality classes is valid. Thus, an or-gate is connected to the single causality class *Class 0*. A causality class is valid when every contained basic event *Car_Ca*, *Car_Cc*, *Train_Ta* and *Train_Tc* occurred, which means that an action trace is obtained from the system that contains exactly these events in a given order. The basic events can occur in the model in different orders. These basic events are combined by an and-gate that requires all of them to occur without imposing a particular order on the occurrence. This fault tree returned by QuantUM does not currently depict these different action orders, even though the ordering information is contained in the set of action traces that form the causality class. Assume, in the example above, that the train enters the crossing before the car, then the car will never enter the crossing, as per the model definition, and the hazard state will not be reached.

We describe the necessity of an action to occur before another action to reach a property violation by a dependency relation. For this dependency relation, the following properties hold.

- An action can not depend on itself, otherwise, the action can never occur (*irreflexivity*).
- When an action b depends on another action a, then a cannot also depend on b (*antisymmetricity*). Assume two actions would mutually dependent on another, then these actions could never occur.
- When an action c depends on b and b depends on a then c depends also on a (*transitivity*).

A strict partial order has exactly these properties. It differs from a (general) partial order only in the property of irreflexivity. We use a strict partial order to describe the dependencies of the actions in action traces.

In this paper, we propose an analysis that computes the strict partial orders of actions in action traces and depicts the computed order in a fault tree. In order to depict the action orders, we introduce the ordered-and-gate into the Fault Tree notation. It is depicted as an and-gate labeled with a triangle. It is satisfied when the actions connected to the gate occur from left to right. For the running example, the analysis results in the fault tree depicted in Figure 1(c) which represents all orders of the action traces given in Figure 1(a) that correspond to the causality criteria defined in Causality Checking. The order in the fault tree depicts, for instance, that Tc occurs always after the other actions, and the action Ta is independent of action Ca and action Car_Cc.

Contributions. In this paper, we present an analysis that computes and depicts the order of actions in the action trace set belonging to a causality class. We also implement this analysis in QuantUM.

Structure. In Section 2 we discuss the foundations of our work. In Section 3 we present an algorithm to compute the strict partial order for the action traces of a causality class. We evaluate and compare an implementation of the algorithms in Section 4. In Section 5 we draw conclusions and suggest future developments.

Related Work A Mazurkiewicz trace [DR95] describes a set of traces by a sequence t of actions and a dependence relation D. The D is symmetric which means when (a, b) is in D then (b, a) is in D. Two in t neighboured actions a and b can be reordered when (a, b) is not in D. In contrast, the strict partial order that our analysis computes is antisymmetric. In the context of causality, either a depends on b then b occurs before a in an action trace, or b depends on a then a occurs before b but both dependencies are not possible at once.

Lamport's happen-before describes strict partial orders for messages in an asynchronous system [Lam78]. In contrast, we compute the strict partial order of actions in a set of action traces.

A (strict) partial order is usually depicted by a Hasse diagram which is an undirected acyclic graph where lower vertices connected to vertices above have to happen first [ES13]. In the context of QuantUM, we prefer to use Fault Trees to depict causality classes and the orders that they represent since they are a notation that is well known to engineers of safety-critical systems.

We are not aware of any work that computes a strict partial order for a set of action traces.

2 Preliminaries

The model of a system is given in form of a transition system [BK08]. 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. An execution p of the transition system TS is an alternating sequence of states $s \in S$ and actions $a \in Act : p = s_0 a_1 s_1 a_2 \ldots$ such that $(s_i, a_{i+1}, s_{i+1}) \in\rightarrow$ for all $i \geq 0$. The behavior of a system is described by the executions of the TS. For an invariant property ϕ, a finite execution $s_0 a_0 s_1 \ldots s_{n-1} a_{n-1} s_n$ where $s_n \nvDash \phi$ is called a counterexample.

An action trace $a_0 a_1 \ldots$ is the projection of an execution $s_0 a_0 s_1 a_1 \ldots$ on Act. An action trace set is a set of action traces T where every action trace t in T has the same alphabet $A \subseteq Act$ and every action of the alphabet occurs in t exactly once. Thus, every action trace in an action trace set has the same length $n = |A|$. Notice that a causality class is an action trace set. An action trace t of a TS can contain an action a several times. In this case, we substitute every occurrence a in t with a_i where the index i is the number of occurrences of a up to the current action in t, and add a_i to the alphabet.

A *directed graph* G is a pair (V, E) of a set of vertices V and a set of edges $E \subseteq V \times V$ where $V \cap E = \varnothing$ [CLF05]. A *walk* of G is a finite sequence of states $u_0, u_1, \ldots, u_n$ where for $0 \leq i \leq n$, $u_i \in V$ and for $0 \leq i \leq n-1$, $(u_i, u_{i+1}) \in E$. A cycle is a nontrivial walk $u_0 \ldots u_n$ with $u_0 = u_n$. A vertex v is connected in G to a vertex u when a walk $v...u$ exists. A graph is connected when for every two vertices v and u in V either a walk $v..u$ or a walk $u..v$ exists. A *directed acyclic graph* (DAG) is a directed graph without a cycle. A tree is an acyclic connected graph [CLF05]. The transitive closure (V, E^*) of a directed graph (V, E) contains an edge (v, u) in E^* for every walk $v...u$ in (V, E).

In concurrent systems, the order in which events can occur is often determined by a *partial order* relation.

Definition 1 (Partial Order [SRH18]). *A homogeneous relation* $\leq \subseteq A \times A$ *is called a partial order over set A if, and only if*

- $\forall a \in A. a \leq a$ *(reflexive)*
- $\forall a, b \in A. a \leq b$ *and* $b \leq a$ *then* $a = b$ *(antisymmetric)*
- $\forall a, b, c \in A. a \leq b$ *and* $b \leq c$ *then* $a \leq c$ *(transitive)*

A *strict partial order* is a partial order that is irreflexive, which means that $\forall a \in A. a \nleq a$ holds. We use the sign $\prec$ to denote a strict partial order.

The function $i_t(a)$ returns the index of an action a in an action trace t. For an action trace set T, we say that an action b depends on a when in every action trace $t \in T$ the index of a is smaller than the index of b. Formally, a depends on b if $\forall t \in T. i_t(a) < i_t(b)$ holds. We express dependencies by a strict partial order. A dependency of action a on action b is denoted by $a \prec b$. When $a \prec b$ holds, we say that a is a precondition for b. When neither $a \prec b$ nor $b \prec a$ holds, we say a and b are independent.

The action set A has a number $|A|$ of actions. For some given causality class over an action set A, we represent the dependencies of the actions in this causality class in a Boolean matrix M of dimension $|A| \times |A|$. An entry (a, b) in M has the value $t$rue when b depends on a in the corresponding causality class.

3 Algorithm to Analyze Action Orders in Action Traces

In this section, we present an algorithm that we refer to as Algorithm 1, which is designed to compute the strict partial order of the actions in a set of action traces. We also define an algorithm called Algorithm 2, which translates this strict partial order into a fault tree.

The input to Algorithm 1 is an action trace set defining a causality class. For instance, in the railroad example the action trace set in Figure 1(a) is the action trace set forming the causality class computed for the railroad model. This action trace set is built over the action set {Car_Ca, Car_Cc, Train_Ta, Train_Tc}. Algorithm 1 computes the strict partial order of the actions in an action trace set and stores it in a DAG which is accomplished in the following way. The strict partial order $a \prec b$ holds for an action trace in which an action a occurs before an action b. In the railroad model, for instance, Train_Tc occurs in every action trace after Car_Cc and Car_Ca. Thus, Car_Cc $\prec$ Train_Tc and Car_Ca $\prec$ Car_Cc holds in every action trace. Since Car_Cc $\prec$ Train_Tc holds in every action trace in Figure 1(a), we deduce that Train_Tc depends on Car_Cc occurring first in order to reach a property violation. In the same way, Car_Cc also depends on Car_Ca and Train_Tc depends on Car_Ca. In the DAG for the railroad example, the algorithm only needs to store the information that

Train_Tc depends on Car_Cc and that Car_Cc depends on action Car_Ca because Train_Tc also transitively depends on Car_Ca. We define the direct dependency relation in Definition 2 that removes transitive dependency relations.

Definition 2 (Direct Dependency Relation $\hat{\prec}$). *An action b directly depends on action a, written as $a \mathrel{\hat{\prec}} b$, in an action trace set T with an action set A when $a \prec b$ holds and $\neg \exists a' \in A. a \prec a' \wedge a' \prec b$.*

For instance, in the running example, the direct dependency relations Car_Ca $\hat{\prec}$ Car_Cc and Car_Cc $\hat{\prec}$ Train_Tc holds and imply the dependency relation Car_Ca $\prec$ Train_Tc but Car_Ca $\hat{\nprec}$ Train_Tc because Car_Cc has a direct dependency with Car_Ca and Train_Tc.

The DAG in which the algorithm stores the dependencies is a DDAG G defined in Definition 3. A DDAG has no superfluous edge e that can be implied by transitivity, formally $(E \backslash e)^* = E^*$, and stores this transitive reduction of the strict partial order.

Definition 3 (Dependency DAG (DDAG)). *A dependency DAG (DDAG) is a DAG (V, E) that stores a strict partial order $\prec$ over a set A with $V = A$ and $(a, b) \in E$ for any actions $a, b \in A$ where $a \mathrel{\hat{\prec}} b$ holds.*

An action trace $t = a_0 \dots a_n$ satisfies G when for every vertex $v \in V$ there exists an action $a_i \in t$, and any two vertices a_i and a_j in t with a walk $a_i ... a_j$ in G satisfy $0 \leq i < j \leq n$.

G is the input for Algorithm 2, which computes a causal tree as defined in Definition 4. The causal tree represents the strict partial order in G. In the context of Causality Checking, a causal tree is the part of a fault tree that represents a single causality class. A causal tree consists of basic events that represent the actions in a causality class, ordered-and-gates where a connected event on the right side depends on every event on the left side, and and-gates where the connected events are independent. The fault tree in Figure 1(c) depicts the basic events Car_Ca, Car_Cc, Train_Ta and Train_Tc. In the fault tree, an ordered-and-gate specifies that Car_Cc depends on Car_Ca to occur first, and a regular and-gate specifies that Train_Ta and Car_Cc are independent.

Definition 4 (Causal Tree). *A causal tree CT is a connected DAG where a vertex v is a basic event for an action, or is a gate. Any vertex v can have an edge (v, g) to a gate g. A gate g is either an ordered-and-gate, where the vertices $v_0 ... v_j$ with edges (v_i, g) are ordered by increasing index i from left to right, or is an and-gate that does not impose an order of the vertices attached to it.*

An action trace $t = a_0 ... a_n$ satisfies the action order imposed by a causal tree CT when every vertex of CT is valid as defined in the following:

- *A basic event v for an action a is valid by t when $\exists 0 \leq i \leq n. t[i] = a$ exists and the validity of v at index i does not contradict the order of an ordered-and-gate in CT.*
- *An order-and-gate og is valid when the vertices $v_0 ... v_j$ become valid in the order $\forall 0 \leq i < i' \leq j. v_i \prec v_{i'}$.*

– *An and-gate g is valid when every vertex v with an edge (v, g) to g is valid.*

In order to compute a fault tree, we compute a causal tree for every causality class and combine the obtained causal trees with an or-gate. For the railroad example, the result of these computations is the fault tree depicted in Figure 1(c). It contains one causal tree which is the subgraph below and including the ordered-and-gate, denoted by the and gate symbol labeled with a triangle.

Algorithm 1 computes the strict partial order of the actions in an action trace set and stores the resulting strict partial order relation in a DDAG. The functions given in Listing 1 compute a DDAG with the strict partial order for a given action trace set `T` built from an action set A. The function `createPreconditionMap` preprocesses the action trace set and returns for any two actions a and b whether the relation $i_t(a) < i_t(b)$ holds in an action trace `t∈T`. These relations are stored in a map `aM` that returns for every action b in A the set of actions that occurred in an action trace directly before b. The function iterates in lines 3 to 5 through every action `t[i]` in every action trace `t` and adds the action `t[i-1]` occurring before `t[i]` to the set of `t[i]` in aM.

The function `createDAG` obtains the map `aM` as an input and computes a DDAG representing the strict partial order of a given action trace set `T`. The algorithm uses `aM` as an input in line 8 to create a dependency matrix `m` of size $|A| \times |A|$. For any actions a and b, an entry (a, b) in `m` is true when $i_t(a) < i_t(b)$ holds in an action trace t of T. Hence, (a, b) is true when an action a is in `aM[b]`. Next, the algorithm ensures that the properties of a strict partial order hold for the relation stored in `m`. In line 10, the algorithm computes the transitive closure of `m` and stores it in `m`. In line 12, the algorithm removes symmetries in `m` by setting (a, b) and (b, a) to false since, as we argue above, symmetrically ordered actions cannot be dependent on each other. In line 13, the algorithm removes reflexive transitions when for an action a the relation $a \prec a$ holds.

```
1   Map<Action, Set<Action>> aM;
2   function createPreconditionMap(Set<ActionTrace> T)
3     for ActionTrace t in T
4       for i: 1 ... t.length - 1
5             aM.get(t[i]).add(t[i-1]);
6
7   function createDAG(Map<Action, List<Action>> aM)
8     Matrix m = createDependencyMatrix(aM);
9     //transitive closure: a1 < a2 && a2 < a3 => a1<a3
10    m = ensureTransitivity(m);
11    //antisymmetric: a1<a2 && a2<a1 => independent(a1, a2)
12    m = ensureAntisymmetricity(m);
13    m = removeReflexivity(m);
14    m = removeTransitiveDependencies(m);
15    return getDAG(m);
```

Listing 1. Pseudocode of Algorithm 1 to Compute DAG.

`m` now contains a strict partial order for `T`. In line 14, the algorithm removes the transitive relations from `m` in order to compute a DDAG that contains only direct dependencies. The algorithm removes transitive relations starting with an action that has the most precondition actions and then iterating in decreasing order over the other actions in A. In order to remove the transitive relations for an action c, the algorithm checks for any actions b and a whether valid entries (b, c) and (a, b) exists in m, in which case it sets the entry (a, c) to false.

In line 15, `m` is converted into a DDAG G. Every action is a vertex in G. For any two actions a and b where the entry (a, b) is valid in m, the algorithm adds an edge (a, b) to the DDAG. This DDAG is returned by the function `getDAG`.

Algorithm 2 uses the DDAG `G` returned by function `getDAG` as an input and computes a causal tree. In lines 4 to 6 in Listing 2, the algorithm first iterates through every action `p` in `G` where `p` is a precondition of another action `a`. It stores this property of p using a Boolean variable Used for `p` in a map `m`. In lines 7 to 9, we search for every action that is not a precondition of another action. These actions are independent of any other action. For every independent action `a`, we call the recursive function `createTree` in line 10 in order to create a tree that represents the dependencies of `a`. The function `createTree` creates a tree

```
1  Map<Action, (Tree, Used)> m;
2  Set<Tree> indep;
3  function createCausalTree(DDAG G)
4    for Action a in G
5      for p in a.getPre()
6        m(p).Used = true
7    for Action a in G
8      if m(a).Used
9         continue;
10     indep.add(createTree(a));
11   return and(indep);
12
13 function createTree(Action a)
14   if(m(a).Tree)
15     return m(a).Tree;
16   Tree t, t';
17   Tree e = createBasicEvent(a);
18   Set<Action> pL = a.getPre();
19   if pL.size() = 0 then t = e;
20   else
21     if pL.size() = 1
22       t' = createTree(pL[0]);
23     else //combine set of preconditions
24       t' = and(foreach p in pL : createTree(p))
25    t = orderedAnd(t', e); //preconditions before e
26    m.put(a, t);
27    return t;
```

Listing 2. Pseudocode of Algorithm 2 to Compute Causal Tree.

for the dependencies of an action `a`. In line 14, the algorithm checks whether the tree of an action `a` was previously created. In case, this tree for `a` is stored in `m`, the function `createTree` returns this tree. Otherwise, the algorithm creates the tree for `a` and stores it in variable `t`. The algorithm first creates a basic event `e` for `a` in line 17. In line 18, the algorithm gets the set `pL` of actions on which `a` depends. In case `a` depends on no other action, `e` is the tree with the dependencies of `a` and the algorithm stores `e` in `t`. In case `pL` contains only a single action stored in `pL[0]`, the algorithm creates the tree `t'` with the dependencies for the action in `pL[0]` in line 22. In case `pL` has several actions, the algorithm creates a tree for every action in `pL` in line 24 and combines these trees with an and-gate `t'`. `g'` depicts the precondition actions for `a`, thus, the algorithm combines `g'` and `a` in line 25 in this sequence with an ordered-and-gate. This ordered-and gate is stored in `t`. `t` is stored in `m` for the action a in line 24 and in line 25 returned by the function. The function `createTree` is called in line 10 for every independent action. The trees of these actions are stored in a set `indep`. After all trees are created, they are combined by an and-gate in line 11 and this and-gate is the causal tree that we wanted to compute.

It is possible to optimize the algorithm in the following way. An ordered-and gate in the causal tree can be connected to another ordered-and gate. For instance, when a_1 occurs before a_2 and a_2 occurs before a_3 then the presented algorithm creates two ordered-and-gates instead of one with all three actions. The implementation of line 22 and 25 in Listing 2 combines several ordered-and-gates to a single one when possible and returns it.

Correctness of the Algorithms. We now prove that the presented algorithms to compute a causal tree that depicts a strict partial order for an action trace set T is correct with respect to completeness and soundness.

A DDAG G computed by Algorithm 1 is complete according to Definition 5 when every action trace in T corresponds to a valid ordering of the actions according to the dependencies stored in G.

Definition 5 (Completeness DDAG Construction). *Assume a DDAG G computed for an action set T. G is complete when any action trace $t \in T$ is an action trace satisfying G.*

Theorem 1 (Completeness of Algorithm 1). *Algorithm 1 computes a complete DDAG according to Definition 5.*

Proof. Assume a DDAG G computed by Algorithm 1 for an action trace set T and an action trace t in T that is not satisfying G. Since t is not satisfying G two actions a and b exist that satisfy $i_t(b) < i_t(a)$ but in G the dependency relation $a \prec b$ holds. Since $a \prec b$ holds in G by construction of G another trace t' in T exists that satisfies $i_{t'}(a) < i_{t'}(b)$. For t and t', Algorithm 1 would store the relations $i_t(b) < i_t(a)$ and $i_{t'}(a) < i_{t'}(b)$ in matrix m (line 8) and removes them (line 12) afterwards since these relations contradict antisymmetricity. Thus, either $a \prec b$ cannot hold in G or $t \notin T$. Both cases contradict our assumptions. □

A DDAG G computed by Algorithm 1 could be considered sound when any action trace that satisfies G is in T. However, as we shall see, this definition of soundness is too strict. Assume, a set with two action traces a, b, c and c, a, b. Then, Algorithm 1 computes a strict partial order $a \prec b$. This strict partial order allows the action trace $t_3 = a, c, b$ but t_3 is not in the original action trace set. This observation was considered further in [Wei19]. For G, we therefore use a different soundness criterium based on pairs of actions. Notice that in an action trace that satisfies G, only the order of independent actions can be changed while preserving its satisfaction of G. As mentioned above, two actions a and b are independent when neither $a \prec b$ nor $b \prec a$ holds in G. $a \prec b$ does not hold when a trace t with $i_t(b) < i_t(a)$ exists, and $b \prec a$ does not hold when a trace t' with $i_{t'}(a) < i_{t'}(b)$ exists. G is sound according to Definition 6 when for any two independent action in G the action traces t and t' exist.

Definition 6 (Soundness DDAG Construction). *Assume a DDAG G computed for an action set T. G is sound when for any two independent action a and b in G, an action trace $t \in T$ satisfying $i_t(b) < i_t(a)$ exists and another action trace $t' \in T$ satisfying $i_{t'}(a) < i_{t'}(b)$ exists.*

Theorem 2 (Soundness of the Algorithm 1). *Algorithm 1 computes a sound DDAG according to Definition 6.*

Proof. Assume a DDAG G computed by Algorithm 1 for an action trace set T and two actions a and b that are independent in G and $a \neq b$. Two actions are independent in G when no walk $a...b$ and no walk $b..a$ exists. By construction of G, a walk $a...b$ does not exist when a trace t with $i_t(b) < i_t(a)$ and a walk $b...a$ does not exist when a trace t' with $i_{t'}(a) < i_{t'}(b)$ exists. We now show by contradiction that the action traces t and t' are in T.

In a first case, we assume that no action trace t exists that satisfies $i_t(b) < i_t(a)$. Thus, the relation $a \prec b$ is not removed in line 12 in Listing 1. In this case, either $a \hat{\prec} b$ and the algorithm creates an edge (a, b) (line 15) or actions a_1, ..., a_n with $a \prec a_1 \prec ... \prec a_n \prec b$ exists and the algorithm creates edges $(a, a_1)(a_1, a_2) \dots (a_n, b)$ in G. Both, the single edge and the sequence of edges represents a walk $a...b$. This walk contradicts the assumption that a and b are independent.

In a second case, we assume that no action trace t' exists that satisfies $i_t(b) < i_t(a)$. This case is equivalent to the first case since a and b are only substituted with another. Thus, the reasoning that t' has to exist is similar to the argumentation for t.

We see that every case contradicts its assumption. Thus, when a and b are independent then t and t' have to exists. □

We now discuss whether Algorithm 1 terminates. Algorithm 1 iterates over actions and their relations. Since the number of action traces in T is finite, the actions and the action relation are also finite. We conclude that Algorithm 1 terminates.

Assume that Algorithm 2 computes a causal tree CT for a DDAG G. Algorithm 2 is sound when for any two actions a and b where $a \prec b$ holds in CT,

$a \prec b$ holds in G, and the algorithm is complete when $a \prec b$ holds in G then $a \prec b$ holds in CT. Remember that action b depends on a does not imply that b directly depends on a, formally $\neg\forall a, b.\ a \prec b \Rightarrow a \hat{\prec} b$. In G, $a \prec b$ holds when a walk $a \ldots b$ exists. In CT, the dependencies of actions are depicted by ordered-and-gates. $a \prec b$ holds in CT when an ordered-and-gate g_b with edges (v_x, g_b) and (b, g_b), where $v_x \prec b$, and a walk $a...v_x g_b$ exist. Definition 7 ensures that an action b depends on an action a in G iff b depends on a in CT.

Definition 7 (Correctness of Causal Tree Construction). *Assume a causal tree CT computed for a DDAG G with an action set A. CT is sound when any two action $a, b \in A$ that satisfy $a \prec b$ in CT also satisfy $a \prec b$ in G. CT is complete when any two action $a, b \in A$ that satisfy $a \prec b$ in G also satisfy $a \prec b$ in CT. CT is correct when it is sound and complete.*

Theorem 3 (Correctness of Algorithm 2). *Algorithm 2 computes a correct causal tree according to Definition 7.*

Proof. Assume a causal tree CT computed by the Algorithm 2 for a DDAG G, and two actions a and b in G. In a first case $\Rightarrow$, we assume that $a \prec b$ holds in G and will show that $a \prec b$ holds in CT, and in a second case $\Leftarrow$, we assume that $a \prec b$ holds in CT and will show that $a \prec b$ holds in G. In line 25 of Listing 2, an ordered-and-gate g_b is created for b when b directly depends on at least one other action in G. Thus, when b depends on another action, g_b exists and when g_b exists, b depends on another action. By the construction of g_b, b is its most right vertex and so for any walk $a..v_x g_b$ in CT, $v_x \prec b$ holds.

- $\Rightarrow$ We assume that $a \prec b$ holds in G but not in CT. Because $a \prec b$ holds in G, a walk $a_0...a_n$ with $a_0 = a$ and $a_n = b$ in G has to exist. This walk witnesses that every action a_i with $0 < i \leq n$ has a precondition. Thus, Algorithm 2 creates an ordered-and-gate for every a_i with $i \geq 1$(line 25 in Listing 2), and for $i > 1$ either an edge (g_{i-1}, g_i) when a_i has a single precondition (line 22), or creates an and-gate g'_i and the edges (g_{i-1}, g'_i) and (g'_i, g_i) (line 24). We see a walk $g_1...g_n$ has to exist in CT. Action a can also have a precondition then similar to the other actions a walk $a g_0 g_1...g_n$ exists in CT. Otherwise, a has no precondition (line 19) and a walk $a g_1...g_n$ exists. Since $g_n = g_b$ both walks $a g_0 g_1...g_n$ and $a g_1...g_n$ witness that a walk $a..g_b$ exists in CT. We conclude that $a \prec b$ holds in CT. This contradicts the assumption that $a \prec b$ does not hold in CT.
- $\Leftarrow$ We assume that $a \prec b$ holds in CT. Thus, an ordered-and-gate g_b with edge (b, g_b) and a walk $a...g_b$ exists in CT. We now construct a walk $a...b$ in G. In CT, an edge (a, g) is either an edge from a basic event to a gate or from a gate to another gate. Hence, a is the only basic event in the walk $a...g_b$ and we know that the other vertices g_0, ..., g_b are gates. Every gate g_i in $g_0...g_b$ is an and-gate or an ordered-and-gate. By construction (line 25 and 17), every ordered-and-gate g_i is created for an action a_x in G and has an edge (g_i, a_x) in CT. An and-gate g_i is created (line 24) when a_x has several preconditions and depicts independence. We can remove every and-gate and

substitute every ordered-and-gate g_i with its action a_x in $ag_0...g_b$ and result in a walk $aa_1...b$. Thus, we found a walk $a...b$ in G that ensures $a \prec b$ in G.

Since both cases hold, we conclude that $a \prec b$ holds in G iff $a \prec b$ holds in CT. □

Algorithm 2 executes only finite loops over the actions in G in the function *createCausalTree* and creates at most once a dependency tree for every action in G. Since the actions in G are finite, Algorithm 2 will terminate.

Theorem 1 and Theorem 2 show that according to our correctness criteria, Algorithm 1 computes a DDAG G with the dependencies contained in an action trace set T. Theorem 3 shows that Algorithm 2 computes a causal tree CT for G that depicts the action dependencies in G. In summary, a causal tree CT computed by Algorithm 1 and Algorithm 2 correctly depicts the action dependencies in T.

Complexity. In the following, we analyze the worst-case complexity of Algorithm 1 and Algorithm 2.

The worst-case complexity of Algorithm 1 is determined by the size $|T|$ of the action trace set T and the size $|A|$ of its alphabet A. Algorithm 1 has several computation steps of different complexity. First, Algorithm 1 iterates over every action trace in T and every action in an action trace (line 3-5), which has a complexity in $O(|A| \cdot |T|)$. In the next computation step in line 8, a lookup is executed for every tuple of two actions in $A \times A$ to create the dependency matrix `m`. Thus, the complexity to create `m` is in $O(|A|^2)$. The worst-case-complexity to compute the transitive closure is in $O(|A|^3)$ [OO73]. For every action in A, irreflexivity is ensured in line 13 and this has a complexity in $O(|A|)$. Next, the transitive dependencies are removed in line 14. Therefore, the actions are ordered by the number of their preconditions, which has a complexity in $O(|A|^2)$ to count the number of preconditions, and a lookup happens for every triple of three different actions in $A \times A \times A$ which results in a complexity of $O(|A|^3)$. In summary, the most complex computation steps are in $O(|A|^3 + |A| \cdot |T|)$ and this is the worst-case complexity of Algorithm 1.

Algorithm 2 first determines the independent actions in G in $O(|A|^2)$. Afterwards, function `CreateTree` is called for every action a in G to depict the dependencies of a. Notice that a depends on at most $|A| - 1$ other actions. For every a, `CreateTree` creates at most one and-gate (line 24), one ordered-and-gate (line 25), and $|A| + 1$ edges. One edge starts in every action on which a depends and one edge starts in every gate that is created. This computation to depict the dependencies of a is executed at most once since the result is stored in the map `m`. We see, `CreateTree` is called $|A|$ times where every call is in $O(|A|))$ which results in an overall worst-case complexity in $O(|A|^2)$. In summary, the worst-case-complexity of Algorithm 2 is in $O(|A|^2)$.

Model	States	Transitions	#Causality Classes	#Traces	#Actions	Time	Memory
Railroad	92	231	1	3	4	4ms	0.605MB
Railroad_gate	143	373	4	20	10	28ms	2.438MB
Airbag	3,456	14,257	5	252	9	28ms	2.622MB
TrainOdometer	4,032	19,624	3	5	5	34ms	2.590MB
FFU_ECU	9,728	30,209	19	80	6	40ms	9.660MB
FFU_Star	207,052	964,695	16	80	6	27ms	17.038MB
ASR	680,897	3,745,635	2	61,920	29	4ms	14.864MB

Table 1. Quantitative experimental results.

4 Case Study

We implemented Algorithm 1 and Algorithm 2 in the tool QuantUM. We qualitatively evaluate the algorithms in that we assess whether they can jointly analyze the strict partial order in a given set of action traces. In the quantitative assessment, we measure the computing resources needed by the algorithms when analyzing a set of models. All experiments were performed on a computer with an i7-6700K CPU (4.00GHz), 60GB of RAM and a Linux operation system.

Qualitative Results and Interpretation. The resulting fault tree of the running example is given in Figure 1(c). In [KL19], we analyzed a slightly different model of the railroad crossing example which includes the functionality of a gate. Without the use of the algorithms proposed in the current paper, QuantUM computes the fault tree in Figure 2 in [KL19] that does not depict the order of the actions. When using the proposed algorithms, QuantUM computes the fault tree in Figure 3. It depicts the order of the actions in a causality class as we defined it above. Both fault trees contain the causality classes Class0 to Class3. In both fault trees, all actions of Class0 are contained in Class2 and all actions of Class1 are contained in Class3. It is not clear from the fault tree in [KL19] why Class2 and Class3 contain minimal counterexamples which is a necessary condition for a cause [KL19]. In the fault tree in Figure 3, we see that in Class0 and Class2 the gate has a failure caused by event Gate_fail. In Class0, the gate is stuck open and in Class2 the gate first closes and then opens in error. Thus, both times the train and the car can be in the crossing at the same time, and therefore incur an accident. In the fault tree in Figure 3, we see the difference between Class0 and Class2 in the order of the actions. This fault tree also depicts the difference between Class1 and Class3. In Class1, the car crosses the railroad and meanwhile, the gate closes and the train enters the crossing. In Class3, two trains enter the crossing subsequently, but the signal gate_open to open the gate is late. Thus, the gate opens when the second train is already in the crossing. The car can then enter, leading to the hazard. In Class1 and Class3 the system behaves without a failure of the system but the order of the actions causes the hazard. We conclude that the ordering of the actions helps to understand causes for the occurrence of the hazards.

Quantitative Results and Interpretation We now want to analyze the performance of the causal tree computation by Algorithm 1 and Algorithm 2. There-

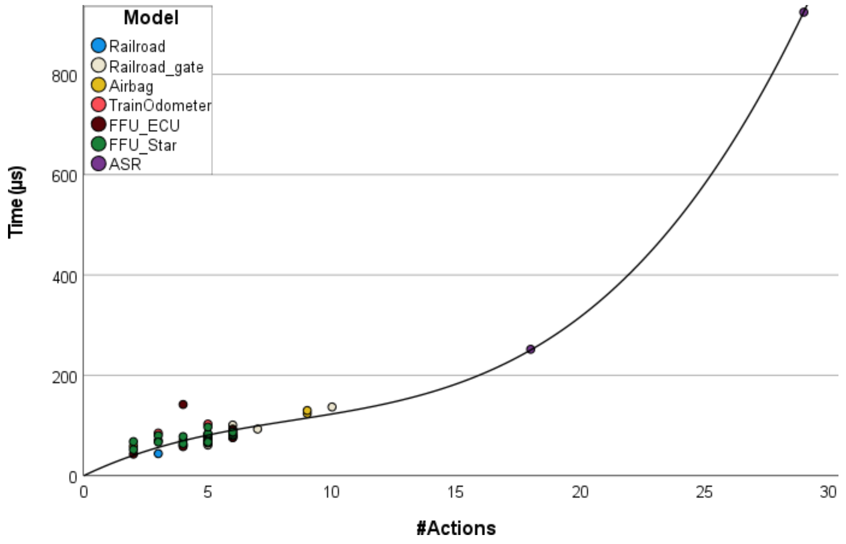


Fig. 2. Time to Compute a Causal Tree in Relation to #Actions in a Causality Class.

fore, we applied the algorithms to several models of different size, in terms of the number of states and transitions that they encompass, taken from [Lei15]. The quantitative results are given in Table 1. The complexity of a model is given in terms of the number of its states and transitions. For every model, we indicate the number of causality classes and the maximal number of traces and actions in one of the causality classes. The columns Time and Memory indicate the maximal computation time and memory consumption that the analysis required in order to compute a fault tree of a model including the action order as per the proposed algorithms.

For every model, QuantUM produces a fault tree where the causality classes are depicted with the strict partial order of the actions. We had a detailed look at all the fault trees and according to this manual inspection, every fault tree depicts the strict partial orders of its causality classes.

The diagram in Figure 2 gives the time in microseconds (μs) that is necessary to compute a causal tree for every causality class in every model. For a model with several causality classes, the diagram depicts several data-points. The worst-case complexity to compute a causality class is the combined worst-case complexity of Algorithm 1 and Algorithm 2 and is in $O(|A|^3)$. We let IBM SPSS [IBM20] analyzed the cubic relation between the time to compute a causality class and the number of actions and IBM SPSS automatically fits the function $31.583 + 13.583x - 1.057x^2 + 0.057x^3$ which is depicted as a black in line in the diagram. The distance of the points to the function can be measured by the coefficient of determination R^2 which is the quadrate of the correlation. The value range of R^2 is $[0, 1]$ where $R^2 = 1$ would be a perfect fit. The function in the diagram has a $R^2 = 0.985$. This function fits nearly perfectly to the data points, which supports that the runtime of the proposed order analysis has a cubic complexity.

While the time to compute a causal tree is given in Figure 2 in microseconds, the overall time to compute a fault tree is in Table 1 in the area of milliseconds. We wondered about this gap of factor 100 and detected that Java, which was used for the implementation of QuantUM and the proposed algorithms, has an offset time in the area of milliseconds to load and create a class when the class is instantiated the first time. This implies that the overall computation times given in Table 1 consists primarily of the time for loading classes and not of the time for computing the causal trees.

Our proposed algorithms computed the strict partial orders within at most 40 milliseconds and at most 17.038MB of memory. This seems reasonable and is acceptable for QuantUM since a causality class in the analyzed models contains up to $61,920$ traces of 29 actions.

5 Conclusion

In this work, we present an algorithm that computes a strict partial order of the actions occurring in an action traces set and represents this strict partial order as a fault tree. We implemented the algorithm in the tool QuantUM and computed fault trees for several models. We showed that a representation of the action order can be computed using a reasonable amount of computing resources, and that the computed results provide helpful insight into the causes for a reachability property violation.

In future research, we plan to further explore the considerations in [Wei19] and to integrate the rewrite-logic based approach pursued in that work with the algorithm described here. Another direction of research is to extend causality checking as well as the computation of event orders in causality classes to the violation of general $\omega-$regular temporal properties.

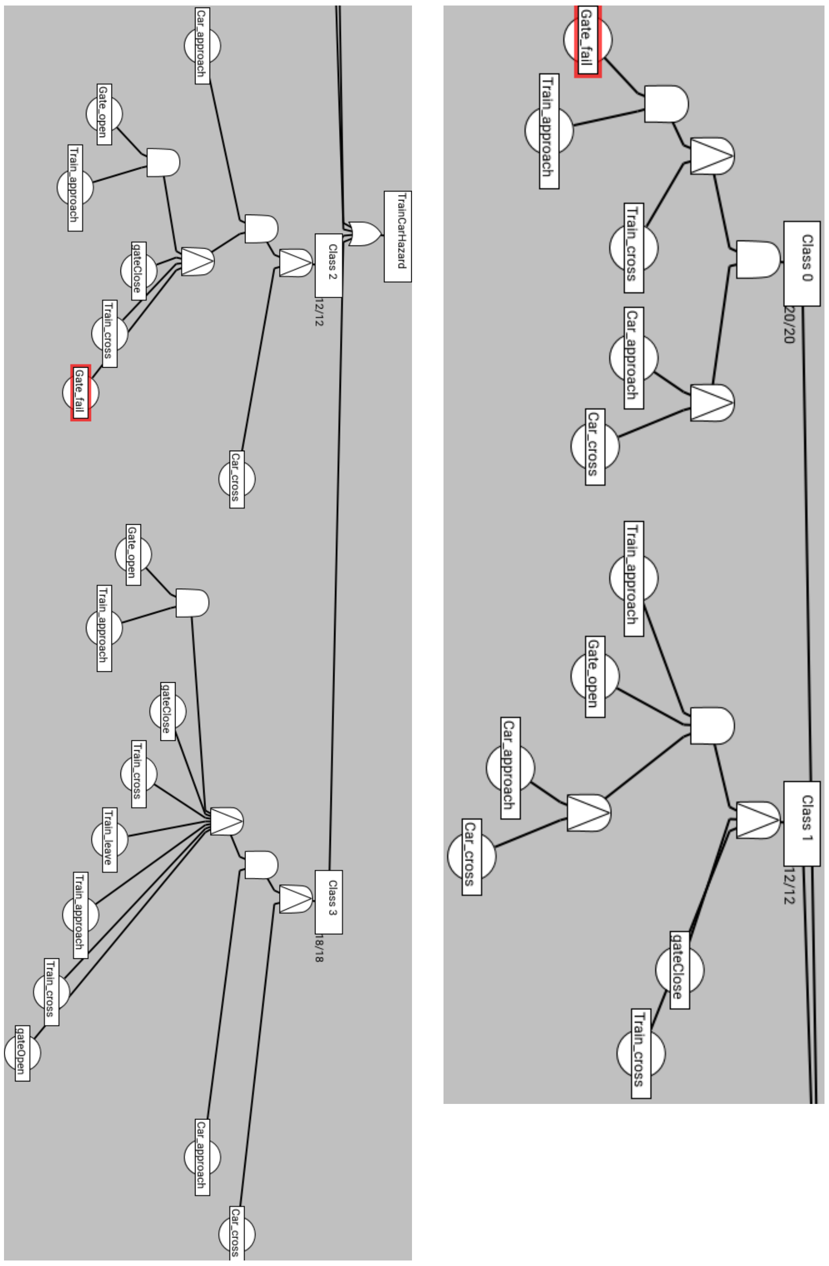


Fig. 3. Fault Tree of Railroad Crossing with Gate with Action Order

References

BK08. Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

CLF05. Gary Chartrand and Linda Lesniak-Foster. *Graphs & digraphs.* Chapman and Hall/CRC, Boca Raton [u.a.], 4. edition, 2005.

DR95. Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces.* World Scientific, 1995.

ES13. David Eppstein and Joseph A. Simons. Confluent hasse diagrams. *J. Graph Algorithms Appl.*, 17(7):689–710, 2013.

Hol04. Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

IBM20. IBM Corp. IBM SPSS Statistics for Windows, Version 27, 2020. `https://www.ibm.com/analytics/spss-statistics-software`.

KL19. Martin Kölbl and Stefan Leue. An efficient algorithm for computing causal trace sets in causality checking. In *ATVA*, volume 11781 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2019.

Lam78. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

Lei15. Florian Leitner-Fischer. *Causality Checking of Safety-Critical Software and Systems.* PhD thesis, University of Konstanz, Germany, 2015.

Lew01. David Lewis. *Counterfactuals.* Wiley-Blackwell, 2001.

LFL11. Florian Leitner-Fischer and Stefan Leue. Quantum: Quantitative safety analysis of uml models. In Mieke Massink and Gethin Norman, editors, *QAPL*, volume 57 of *EPTCS*, pages 16–30, 2011.

LL13. Florian Leitner-Fischer and Stefan Leue. Causality checking for complex system models. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 248–267. Springer, 2013.

Obj17. Object Management Group. OMG Systems Modeling Language, Specification 1.5, 2017. `http://www.omg.org/spec/SysML`.

OO73. Patrick E. O'Neil and Elizabeth J. O'Neil. A fast expected time algorithm for boolean matrix multiplication and transitive closure. *Inf. Control.*, 22(2):132–138, 1973.

SRH18. Bernhard Steffen, Oliver Rüthing, and Michael Huth. *Mathematical Foundations of Advanced Informatics, Volume 1: Inductive Approaches.* Springer, 2018.

Wei19. Jannis Weiser. Derivation of a minimal representation of incomplete partial orders from event sequences. Master's thesis, University of Konstanz, 2019.