An Efficient Algorithm for Computing Causal Trace Sets in Causality Checking

Martin Kölbl and Stefan Leue

University of Konstanz, Germany

Abstract. Causality Checking [LL13a] has been proposed as a finite state space exploration technique which computes ordered sequences of events that are considered to cause the violation of a reachability property. A crucial point in the implementation of Causality Checking is the computation and storage of all minimal counterexamples found during state space exploration. We refer to the set of all minimal counterexamples as a causal trace set. However, the Duplicate State Prefix Matching (DSPM) Algorithm that is currently used in Causality Checking only under-approximates the causal trace set. As we argue, without the approximation the DSPM algorithm is inefficient. We propose the, to the best of our knowledge, first efficient algorithm that precisely computes a causal trace set, avoiding approximation, called Causal Trace Backward Search (CTBS). We compare the DSPM and CTBS algorithms with respect to their worst case complexities, and by applying them to several case studies.

1 Introduction

Causality Checking [LL13a] has been proposed as a finite state space exploration technique which computes sets of minimal ordered sequences of events that are considered to be causal for the violation of a reachability property. The notion of causality used in Causality Checking is an adaptation of the counterfactual causal analysis proposed in the seminal work by Halpern and Pearl on actual causation [HP05, Hal15] to a trace-based model of computation. The sets are referred to as causality classes and are computed in an automated fashion. The union of all causality classes, corresponding to their logical disjunction, is referred to as an actual cause. A causal trace is minimal when none of its non-contiguous subtraces leads to a property violation.

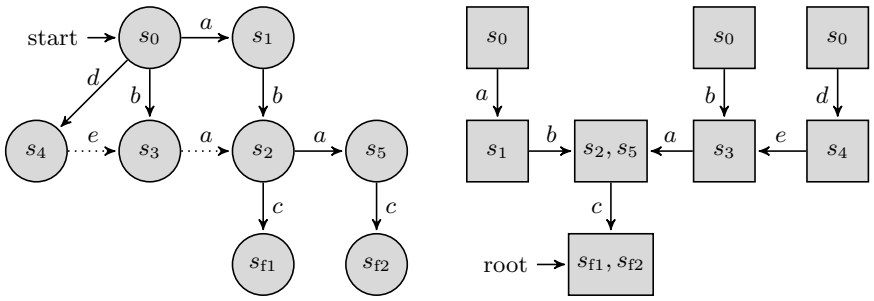
Causality Checking has been implemented in the SpinCause tool [LL14], which computes actual causes for SPIN [Hol04] models. More comprehensively, the QuantUM tool [LL11] implements Causality Checking in order to compute causes for the reachability of hazardous system states in SysML [Obj17] models. QuantUM represents actual causes as formulae in event order logic [LL13b] and visualizes them in the form of Fault Trees [VGRH02], which can then be used as evidence in safety cases for safety-critical systems. We illustrate the application of QuantUM to safety analyses for automotive autonomous driving architectures in [KL18].

In this paper we restrict ourselves to considering hazards that can be detected by reachability analysis on the state space defined by the model that is being considered. When the state space exploration during Causality Checking, typically implemented using a depth-first (DFS) or breadth-first search (BFS), reaches a hazardous state, a trace leading into this state, called a counterexample, will be generated. In implementations of Causality Checking, the state space traversal is usually implemented using a modified model checking algorithm. The modifications concern the fact that we need to explore all executions of the model, including property violating counterexamples and non-violating executions, in order to implement the counterfactual-style actual cause conditions in Causality Checking.

Pivotal for the performance of Causality Checking is the computation and storage of all minimal counterexamples found during state space exploration. Existing implementations of Causality Checking use a prefix tree data structure to store system executions as well as a parallelization of the search in order to improve performance. A crucial aspect in this regard is how the state space exploration deals with the situation in which a state s is visited that has been visited before during the search. s is then referred to as a duplicate state. Both a BFS and a DFS would not further explore a duplicate state, since doing so would lead to an exponential time penalty. However, when performing Causality Checking all executions need to be explored, irrespective of whether they contain a state which, due to the search strategy employed, happens to be a duplicate state. As a consequence, during the exploration of the system executions it is necessary to concatenate the prefixes leading from the initial system state into a duplicate state with all possible suffixes starting with the duplicate state. This should be done efficiently, in particular by behaving benevolently on practical examples in the face of a potential exponential time and space penalty.

For performance reasons the algorithm that is implemented in SpinCause and QuantUM, which we refer to as Duplicate State Prefix Matching (DSPM) and which is documented in [Lei15], under-approximates the computation of the execution suffixes beginning in duplicate states since it cannot guarantee that all of them will be considered. It is the objective of this paper to propose an algorithm that deals with duplicates precisely, without relying on an approximation, and which is nonetheless efficient. The algorithm that we propose, which we refer to as Causal Trace Backward Search (CTBS), performs an efficient exploration and analysis of all counterexamples found during the state space exploration. It works on-the-fly and returns preliminary results at any point during the computation. While DSPM generates valuable approximations for realistic models, as we shall see, it is incomplete for Causality Checking, whereas CTBS is complete.

A Motivating Example. Consider the transition system depicted in Figure 1(a). Causality Checking conceptually works on transition systems, which can be automatically derived from higher-level modeling languages, such as SysML. Assume that states s_{f1} and s_{f2} violate a uniquely defined reachability property φ, corresponding to the occurrence of a hazard in the underlying domain model. We henceforth refer to these states as failure states. In effect, in the context of



(a) Transition System

(b) BCT tree with 3 minimal counterexamples

Fig. 1. Running Example

Causality Checking, reaching a failure state corresponds to the effect for which we compute the actual causes.

The action traces *abc*, *bac* and *deac* are minimal counterexamples for the violation of ϕ. They are minimal since they do not contain a non-contiguous subtrace that also is a counterexample of φ. In this sense, *abac* is a non-minimal counterexample, since it contains *abc* as a non-contiguous subtrace. We refer to the set of all minimal counterexamples as causal trace set. The causality classes defined in Causality Checking will be constructed from the causal trace set by grouping all traces that contain the same set of actions into one such class. In the example, the traces *abc* and *bac* form one causality class.

A standard DFS or BFS, modified to compute all traces, would not return the trace *bac* in case it had previously explored the trace *abc*, since state s_2 would then be a duplicate state. We, hence, need to ensure that a state space exploration concatenates all trace prefixes starting in the initial state and leading into any duplicate state with all trace suffixes that start in the respective duplicate state. In the example this means that the prefix *ba*, for instance, needs to be concatenated with the suffixes *c* and *ac*. Notice that the resulting trace *baac* will not be included in the causal trace set since it is not a minimal counterexample.

As we shall see, the DSPM algorithm under-approximates this concatenation step by not considering all suffixes starting in a duplicate state, in particular when a prefix trace leads into a duplicate state while traversing another duplicate state. Assume DSPM to explore *abc*, *b* and then *de*, which leads to duplicate state s_3. At this point, no suffixes starting in s_3 can be added since none exist. Assume *ba* to be explored next, leading to a second duplicate state s_2. The suffix traces *c* and *ac* would be concatenated to the prefix *ba*. DSPM would in this situation disregard concatenating the suffixes of duplicate states via which s_2 can be reached, such as s_3, and therefore returns an incomplete result by disregarding, for instance, *deac*. Notice that this incompleteness would not occur in case *bac* was explored first, followed by *de*. In this situation DSPM would correctly perform all concatenations of prefixes and suffixes at all du-

plicate states. The algorithm CTBS that we propose in this paper completely handles all concatenations entailed by duplicate states.

Related Work. There are only few papers available that address the computation of all minimal counterexamples for reachability properties required to compute causality classes [LL14, Lei15, BHK+15]. As we argue above, the algorithm described in [LL14,Lei15] only computes an approximation. The algorithm in [BHK+15] is also approximative since it is based on bounded model checking. We propose an algorithm that can in principle compute complete causality classes. Note that minimal traces in a causality class are different from minimal length paths in graphs, which is why the vast literature on shortest path searches on graphs [ES12] and the computation of minimal length counterexamples [AL10, HKD09, SB05, HK06] is not directly applicable to our problem.

Structure of the Paper. We discuss the foundations of our work in Section 2. In Section 3, we present the algorithm proposed in [Lei15] to compute a causal trace set, propose a new algorithm and compare the computational complexity of both algorithms. We qualitatively and quantitatively compare the algorithms by several case studies in Section 4. In Section 5, we draw conclusions and suggest future developments.

2 Preliminaries

Causality Checking uses a modified state space exploration algorithm to traverse the state space of a transition system.

Definition 1 (Transition System (TS) [BK+08]). *A transition system 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.*

The standard state space exploration strategies used in depth-first-search (DFS) or breath-first-search (BFS) are modified for the purpose of Causality Checking in two fundamental ways. First, the state space exploration continues after a first property violating state is found. Second, when reaching a duplicate state, a concatenation operation as explained above needs to be performed.

An *execution* ρ of an TS is a possibly infinite alternating sequence $s_0 a_0 s_1 a_1 \ldots$ of states and actions that starts in the initial state and any triple (s_i, a_i, s_{i+1}), called a transition, is an element in $\rightarrow$. During state space exploration a state s may be visited twice. We then call s a *duplicate state*. A finite execution $s_0 a_0 s_1 \ldots s_n$ of a TS where the last state s_n is a duplicate state is called a *duplicate execution*. A finite execution $\sigma = s_0 a_0 s_1 \ldots s_n$ where $s_n \not\models \varphi$, for an invariant property φ, is called a *counterexample*. We then write $\sigma \not\models \varphi$. An *action trace* $a_0 a_1 \ldots$ is a sequence of actions. In the following, we refer to an action trace simply as a trace.

A trace $\sigma' = a'_0 \ldots a'_n$ contains another trace $\sigma = a_0 \ldots a_m$ when σ is a non-contiguous subtrace of σ', written as $\sigma \sqsubseteq \sigma'$. Formally, $\sigma \sqsubseteq \sigma'$ holds iff the

word $a'_0 \dots a'_n$ is contained in the language obtained from the regular expression $\Sigma^* a_0 \Sigma^* a_1 \Sigma^* \dots \Sigma^* a_m \Sigma^*$, where $\Sigma = \{a'_0 \dots a'_n\}$. We write $\sigma \sqsubset \sigma'$ iff $\sigma \sqsubseteq \sigma'$ and $\sigma \neq \sigma'$. Let $\eta = s_0 a_0 \dots a_m s_{m+1}$ and $\eta' = s'_0 a'_0 \dots a'_n s'_{n+1}$ executions from which traces σ and σ', respectively, have been derived by projection. We say η' contains η iff $\sigma \sqsubseteq \sigma'$. The $\sqsubseteq$ relation is transitive.

For space efficiency reasons, traces are stored using a prefix tree [Fre60] data structure. For any tree, the path of some vertex is defined as a backwards sequence of edges that leads from the considered vertex to the root vertex r. Note that the path of any vertex is unique. A trace of a path is the projection of the respective path on the set of actions. A path p contains another path p' iff the trace of p contains the trace of p'.

3 Algorithms for Computing a Causal Trace Set

Definition of Causal Trace Set. The definition of the causal trace set relies on two essential properties of the traces included in the set. First, every counterexample needs to be represented by an element of the Causal Trace Set (completeness), and second, the causal trace set contains only traces corresponding to minimal counterexamples.

Definition 2 (Causal Trace Set). *Assume a TS T and a safety property φ. Let σ and σ' be traces in T. A causal trace set is a subset Ψ of the traces of T that satisfies following conditions:*

- *TC1 (completeness): For every σ' that satisfies $\sigma' \not\models \varphi$ there exists a σ such that $\sigma \in \Psi$ and $\sigma \sqsubseteq \sigma'$.*
- *TC2 (minimality): Ψ is minimal, that is to say, no $\sigma \in \Psi$ is a true subtrace of $\sigma' \in \Psi$.*

We call a trace in Ψ a causal trace.

Duplicate State Prefix Matching Algorithm (DSPM). We now discuss the DSPM algorithm in more detail. When the state space exploration encounters a counterexample or a duplicate execution, it hands the execution over to the DSPM algorithm. The DSPM algorithm compares new with existing counterexamples and only stores minimal ones. Duplicate executions are stored in a list until the state space exploration terminates. They are then concatenated with previously stored minimal counterexamples which contain the respective duplicate state. The DSPM pseudo code can be found in [Lei15].

As discussed above, DSPM computes a potentially incomplete causal trace set when it is possible to reach a duplicate state via another duplicate state. This is due to the fact that the duplicate processing happens only after all counterexamples have been computed. The order of processing of duplicates depends on the search order used by the state space exploration algorithm. As explained above, the order of encountering duplicate states and their processing may lead to an incompleteness in the discovery of causal traces. It is not obvious

whether there is an ordering that would avoid this incompleteness. In particular, ordering the processing of duplicate states according to the length of the trace needed to reach them will not solve the problem, as we found out.

Causal Trace Backward Search Algorithm (CTBS). The CTBS algorithm computes minimal counterexamples using a *Backwards Causal Trace* (BCT) tree data structure. Consider the example BCT tree depicted in Figure 1(b) which is derived from the transition system in Figure 1(a). CTBS is interleaved with the state space exploration algorithm. When the state space exploration encounters an execution corresponding to a counterexample, to reaching a non-property violating trace ending in a terminal state or to reaching a duplicate state, the corresponding execution will be handed over to CTBS. CTBS maintains the BCT tree data structure, which is implemented as a prefix tree. The edges of the BCT are labeled with actions. The root vertex of this tree is labeled with all failure states of the system, in the example with the states s_{f1} and s_{f2}. The non-root vertices are labeled with a set of states. These states are equivalent in the sense that one can reach one of the failure states from them via an identical trace. The trace is defined by the sequence of action labels along the edges of the BCT tree that are encountered on the path from the considered vertex to the root vertex. As an example, the vertex labeled s_2, s_5 implies that from states s_2 and s_5 a failure state can be reached via a trace c. Notice that such traces correspond to suffixes of counterexamples. When a suffix contains an initial state, then it represents a causal trace. As an example consider the trace *abc* leading from the leaf vertex s_0 to the root vertex.

The CTBS algorithm is designed to satisfy two major requirements: 1) It needs to ensure that all traces in the system are completely analyzed, independently of the search order during the state space exploration. 2) The algorithm should be efficient wrt. both space and time, in particular by storing only minimal counterexamples and by ignoring non-minimal counterexamples.

CTBS computes traces as follows. Assume the BCT tree to be labeled by all failure states. When CTBS receives an execution, it will first split the execution into the transition triples (s_i, a_i, s_{i+1}) that it is built of. In the sequel we will refer to s_{i+1} as the target state of that transition. For each of these transition triples we ensure that there is a child vertex labeled by s_{i+1} for a father vertex labeled s_i in the BCT tree. If the child vertex does not exist, it will be added to the tree. The edge leading to this child will be labeled with a_i. If the child vertex exists, but the edge to the child is labeled by some $a_j \neq a_i$, then a new child node labeled s_{i+1} will be added and the edge leading to this note will be labeled with a_i.

In order to ensure efficiency, the algorithm exploits the observation that for each state s in a minimal counterexample, there is no shorter trace to reach a failure state from s than the suffix of the trace corresponding to the counterexample that starts in s. This implies that all non-minimal suffixes can be removed from the BCT tree, as expressed by the prune rules *PR1* and *PR2*. To illustrate this point, assume that a given BCT tree contains a path *bac* and that the state space exploration hands the trace *abac* over to CTBS. Assume further that *abac*

will be split into transition triples and integrated into the BCT tree. Since *bac* is contained in *abac*, the vertex that stores *abac* and the complete subtree that hangs off it will be pruned from the BCT tree.

> *PR1.* The CTBS algorithm deletes a vertex when the trace of the vertex contains a shorter counterexample.

Lemma 1 shows that the trace of a vertex deleted by prune rule PR1 is not a suffix of a minimal counterexample.

Lemma 1. *A suffix that contains a shorter initial trace is not a suffix of a minimal counterexample.*

Proof. Assume a trace t and a shorter initial trace t' that satisfies $t' \sqsubset t$, and an arbitrary minimal counterexample t_c with suffix t. t being a suffix of t_c implies that $t \sqsubseteq t_c$. $t \sqsubseteq t_c$ and the assumption $t' \sqsubset t$ interrelate transitively, which yields $t' \sqsubset t_c$. t' is a counterexample since it is initial. As consequence, any counterexample t_c is not minimal since t' is minimal. □

A second prune rule removes a state label from a vertex when another vertex labeled with this state has a shorter trace to reach a failure state. Assume a vertex labeled with s_2 to be in the BCT tree of Figure 1(b) from which the root vertex can be reached via trace *ac*. The BCT tree contains a vertex labeled s_2s_5 which has a trace *c*. The trace of vertex s_2 contains the trace of vertex s_2s_5. As a consequence, prune rule PR2 removes the state label s_2 from the respective vertex and thereby removes the longer suffix *ac* from the BCT tree.

> *PR2.* The algorithm CTBS removes a state s from a vertex v whenever the trace t_v of v contains the trace of another vertex v' labeled with s.

Lemma 2 shows that a counterexample with a suffix that is equivalent to the trace of the vertex v in *PR2* cannot be minimal.

Lemma 2. *Any counterexample with a suffix starting in a state s in vertex v is not minimal when another vertex v' with state s exists and the traces t_v contains trace $t_{v'}$.*

Proof. Assume two different vertices v and v' with state s and with traces t_v and $t_{v'}$ that satisfy $t_{v'} \sqsubseteq t_v$, and an arbitrary counterexample c with suffix $t_{v'}$. The counterexample c is not minimal since we can construct a shorter counterexample than c. Notice that $t_v \neq t_{v'}$ otherwise $v = v'$. We split c in state s in a prefix $s_0a_0 \ldots s$ and the suffix $t_v = s \ldots s_f$, prepend the prefix to the suffix $t_{v'} = s \ldots s_{f'}$ and result with a counterexample $c' = s_0a_0 \ldots s \ldots s_{f'}$. The assumption $t_{v'} \sqsubseteq t_v$ with $t_v \neq t_{v'}$ results in $t_{v'} \sqsubset t_v$. $c' \sqsubset c$ holds because the prefixes of c and c' before state s are equivalent by construction and for the suffixes $t_{v'} \sqsubset t_v$ holds. Thus, c' is shorter than c and c is not minimal. □

The pseudo code of the CTBS algorithm is shown in Listing 1.1. The algorithm uses four data structures: The vertex **root** is the root vertex of the prefix tree, a state **initial** stores the initial state of the TS, a map **v_map** returns vertices that contain a certain state, and a map **t_map** returns a list of transitions with a certain state as the target state.

```
1  Vertex root; State initial;
2  Map<State, Set<Vertex>> v_map;
3  Map<State, Set<Transition>> t_map;
4
5  function addExecution(Exection e)
6     IF e.hasBad()
7         //add property violating states to root
8         root.addState(e.lastState());
9     initial = e.firstState();
10    //iterate execution transitions
11    FOR Transition t in e
12        t_map.get(t.s2).add(t);
13        addTransition(t.s1, t.act, t.s2);
14
15 function addTransition(s1, act, s2)
16    //get all vertices with state s2
17    FOR Vertex v2 in v_map(s2)
18       //get father vertex reachable by label act
19       Vertex v1 = v2.getFather(act)
20       //ensure restriction PR1 and PR2
21       IF causalPathShorter(v1) || otherShorter(v1, s1)
22          continue;
23       v1.add(s1);
24       checkOtherLonger(v1, s1); //enforce PR2
25       IF s1 == initial
26          checkAllPaths(v1);     //enforce PR1
27          return;
28       FOR Transition t' in t_map.get(s1)
29          addTransition(t'.s1, t'.act, t'.s2);
```

Listing 1.1. Sketch of Backward Causal Trace Search Algorithm

The algorithm calls the function **addExecution** iteratively for a counterexample or a duplicate execution. If the function is called with a counterexample, the last state of the execution violates the property and this state is added to the root (lines 6-8). For counterexamples and duplicate executions, the algorithm saves the initial state which is the same state for all executions of a TS in variable **initial** (line 9).

The algorithm adds the transitions belonging to an execution to the **t_map** and calls, for every transition $t = (s_1, a, s_2)$, the function **addTransition** (line 11-13). This function checks for every vertex v with state s_2 (line 17) whether a vertex v_1 with state s_1 and edge (v_1, a, v) (line 19) does satisfy one of the prune rules PR1 and PR2 (line 21), and in this case continues with the next

vertex (line 22). Otherwise, both prune rules are not satisfied, s_1 is part of a new minimal suffix and the state s_1 is added to v_1 (line 23). The algorithm then checks whether any other path of a vertex with state s_1 now contains the path of v_1 (line 24) and removes s_1 from such a vertex. If s_1 is an initial state, then a causal trace is found and all other paths in the tree are checked to determine whether they contain the new causal trace (line 26). The transitions of a prefix that reaches state s_1 can already be contained in the transition list **t_map** and need again to be added to the BCT tree. Therefore, the function calls itself recursively for all transitions t' with s_1 as a target state (line 28-29). In order to ensure a clear presentation, we removed several optimization from the pseudo code. Most importantly, in order to optimize performance **addExecution** calls **addTransition** only for transitions not yet in **t_map**. Furthermore, if v_1 in the function **addTransition** already contains the state s_1, than this vertex is ignored and the recursion is not called.

Correctness of the CTBS algorithm. First, it should be noted that CTBS delivers a sound BCT tree since every trace in the computed BCT tree corresponds to an execution of the TS that we consider. The CTBS algorithm is correct iff the set of traces that the BCT tree represents is a causal trace set, in other words, if this set satisfies conditions TC1 and TC2 from Definition 2.

Lemma 3. *The set of traces represented by the BCT tree computed by the CTBS algorithm satisfies condition TC1 of Definition 2.*

Proof. Assume a set Ψ of traces computed by the CTBS algorithm for a set of executions that is returned by a state space exploration algorithm for a TS S, and a counterexample t that does not contain any trace in Ψ. By the construction of Ψ, all traces in Ψ are contained in a BCT tree PT. Obviously, t is not a trace of the tree, otherwise, t itself is in Ψ. t is not a trace in the BCT tree in two cases.

1. t was never added as a trace to PT. This means that there is in the sequence of transitions that corresponds to trace t at least one transition (s_i, a_i, s_{i+1}) so that s_i is not a target state of any other transition in PT. This means that this transition was never handed over by the state space exploration algorithm. A state space exploration algorithm explores the full state space of an S and every transition of S is contained in at least one execution. Since t contains a transition that is not contained in S, t is not a trace of S. This contradicts the assumption that t is a counterexample of S.
2. t was deleted by a prune rule because t contains a shorter counterexample t'. In that case, t' is a trace in Ψ. Otherwise, t' is also deleted because of an even shorter counterexample that is contained in PT.

Either t does not exist or contains a counterexample in PT. Both cases contradict the assumption that a counterexample t exists that contains no trace in Ψ. □

Notice that this completeness result also holds for the CTBS algorithm when it is used with a BFS algorithm for the state space exploration. Assume that the

completeness does not hold. Then there would be a counterexample of length shorter than the current search depth that contains an unexplored transition, which means that the corresponding trace is not included in the BCT tree PT. This, however, contradicts the property of a BFS that all states up to the current search depth have been explored. This property is beneficial in case we need to bound the search depth in causality checking for very large models since it ensures that the causal traces up to the current search depth form a causal trace set, which implies completeness up to the search depth reached.

Lemma 4. *The set of traces in the BCT tree computed by the CTBS algorithm contains only minimal counterexamples.*

Proof. Assume a BCT tree PT computed by the CTBS algorithm for a TS S, and a vertex with an initial state s_0 and a non-minimal trace t contained in PT. t is a non-minimal trace if another counterexample t' exists in S with $t' \sqsubset t$. There are two cases where t' can exist. In the first case, t' is not contained in PT and this case violates Lemma 3. In the second case, t' is contained in PT. In this case a vertex v with the initial state s_0 exists for t. Also, for t a corresponding vertex v' exists. One of the vertices already contains s_0 and with adding s_0 to the other vertex, the corresponding traces t and t' are compared by prune rule PR1 when determining whether the vertices of t or t' can be removed. This leads to two cases. If v already contains s_0, then s_0 is removed from v. Otherwise, if v' already contains s_0, then s_0 is removed from v. In both cases, v does not contain s_0. This contradicts the assumption that t is an initial trace. □

The correctness of CBTS follows from Lemmas 3 and 4.

Complexity Considerations. The *worst-case size of the causal trace set* is bounded by the number of traces in a TS, since all traces can be minimal traces. A minimal trace can be an arbitrary sequence of actions from the action set Act, where the corresponding execution of the TS contains any state at most once. Traces generated by loops in the TS do not need to be considered in the complexity analysis, as can easily be seen. Assume a minimal trace that reaches a state twice, then the subtrace between reaching the state for the first and the second time corresponds to a loop in the TS. A trace not including this loop is shorter and reaches the same end state as the trace that contains the loop, which means that the trace including the loop cannot be minimal. In the worst case, the number of traces in a TS is $|Act|^{n-1}$, where $|Act|$ is the number of actions and n is the number of states in the TS.

The *worst case complexity of the DSPM algorithm* was shown in [Lei15] to be in $\mathcal{O}(|t|^2)$ where $|t|$ is the number of all traces in an TS. The complexity is driven by the comparison of every trace in t with all causal traces, which in the worst case can be all traces.

The *worst case complexity of the CTBS algorithm* is also dominated by the number of trace comparisons. The algorithm processes iteratively the set of transitions E of a TS. The number of transitions $|E|$ is at most $|t|$ in the worst case in which all traces t consist of a single transition. A transition can only be added

once to a trace in the BCT tree. Otherwise the trace would not be minimal since it would contain the target state of the transition twice. In the worst case a transition is added to all traces in the BCT tree. When adding a state to a vertex, the prune rules need to compare the trace of this vertex in the worst case twice with all other traces. We conclude that the CTBS algorithm has a worst case complexity of $|E| \cdot |t| \cdot 2 \cdot |t| \in \mathcal{O}(|t|^3)$.

In conclusion, under worst case complexity considerations, DSPM scales better than CTBS. The approximation performed by the DSPM algorithm does not affect the worst case runtime since the trace set t consists of all traces, including the traces not analyzed by DSPM.

The worst case complexity of computing a causal trace set depends on the number of all traces that need to be computed, which as shown above can be exponential. However, for realistic models not all traces are minimal, and hence not causal. This means that there is significant potential in removing non-causal traces. The DSPM algorithm always analyzes complete counterexamples. In contrast, due to the prune rules the CTBS algorithm compares and prunes suffixes before a complete counterexample is analyzed. This is the essential performance advantage of the CTBS algorithm over DSPM, as will become obvious during the experimental evaluation.

4 Case Study and Experimental Evaluation

We now compare the DSPM and CTSB algorithms by analyzing several models of different size taken from [Lei15]. We implemented four variants of the considered algorithms:

(1) the CTBS algorithm based on BFS state space exploration,
(2) the CTBS algorithm based on DFS state space exploration,
(3) a modified version of the DSPM algorithm, and
(4) to obtain a baseline, an algorithm that ignores all duplicate states, i.e., behaves like a standard BFS when encountering a duplicate state.

These implementations are integrated in the `QuantUM` tool and use a modified version the model checker SpinJa [dJR10] for the state space exploration. We modified the above described DSPM algorithm in order to address its insufficiencies with respect to treating multiple duplicates encountered during state space exploration. This is accomplished by computing all minimal counterexamples by repeatedly iterating through all duplicate states stored in a list maintained by DSPM until no new counterexample is generated.

All analyses were performed on a computer with a E5-2697 CPU (2.7GHz), 785GB of RAM and a 64 bit Linux operating system. The analysis algorithms were mapped to two threads: one thread performed the state space exploration and another executed the algorithm for the causal trace set computation. The state space exploration was based on a BFS as long as nothing else is stated. A timeout for the experiments was set to two hours.

Model	States	Transitions	Depth	#Causal	#Class	$Time_{Last}$	$Time_R$	Memory
Railroad crossing	143	373	37	62	4	<00:01	00:04	51MB
Airbag	3,456	14,257	35	484	5	<00:01	07:09	62MB
Odometer	4,032	19,624	55	13	3	<00:01	00:09	70MB
FFU Star	207,052	964,695	37/60	458	16	<00:01	timeout	545MB
FFU ECU	235,765	90,775,575	31/60	509	19	<00:01	timeout	896MB
ASR ch1	680,897	3,745,635	37/60	67200	2	02:00:05	timeout	3,747MB
ASR Reduced	14,222,115	45,997,298	39/60	76428	2	01:45:58	timeout	91,814MB

Table 1. Experimental results for CTBS algorithm (variant 1).

Table 1 shows the results of the CTBS algorithm. The input model is characterized by the number of states that were explored, the number of transitions traversed and the maximal search depth. For the larger models `FFU Star`, `FFU ECU`, `ASR ch1` and `ASR Reduced` we limited the search depth to 60. In case of a timeout, the current search depth reached by the CTBS algorithm is given in column *Depth*. For all of the four larger models, the search depth limit was reached when the timeout occurred. We indicate time values using the format (hours:minutes:seconds). The column $\#Causal$ gives the size of the causal trace set that was computed, $\#Class$ gives the number of causal classes that were detected, and $Time_{Last}$ indicates the period of time after starting the experiment when the last causal trace was found. The computation time consumed by the analysis is given in the $Time_R$ column, and the consumed memory in the column *Memory*. Table 2 shows some experimental results for the algorithm variants 4), 3) and 2).

In order to compare the algorithm variants qualitatively, we analyzed the railroad crossing example taken from [Lei15] with the CTBS variant 1) using the QuantUM tool. We obtained the fault tree depicted in Figure 2. The fault tree has four subtrees that represent the causality classes `Class 0` ... `Class 3`. A class is created by all causal traces with the same set of actions. The number of causal traces that a class contains is depicted next to the class name, for instance, `Class 0` contains 20 causal traces. These causal traces lead to a partial order representing 20 different linear orderings of the 5 class events in `Class 0`. Notice that the type of fault tree that we use is a dynamic fault tree, which means that the order of the occurrence of events on one of its and-branches is assumed to be linear from the top to the bottom. In effect, Causality Checking computes a partial order of these events that cannot be directly depicted in a fault tree. To describe these partial order constraints we use event order logic, and we omit these ordering constraints here.

Quantitative Result Interpretation. The algorithm variant 4) defines a base line for the quantitative performance of the other algorithm variants when analyzing the different models. The highest analysis time with this algorithm is observed for the model `ASR Reduced` with a value of 57:57. The algorithm variants 1), 2) and 3) have a much higher computation time demand. As opposed to the baseline, these algorithms experienced a timeout for the models `FFU Star`, `FFU ECU`, `ASR ch1` and `ASR Reduced`. We can conclude that the penalty for processing duplicate

Model	Ignore Duplicate Traces			mod. DSPM algorithm			CTBS with DFS		
	#Causal	#Class	$Time_R$	#Causal	#Class	$Time_R$	#Causal	#Class	$Time_R$
Railroad crossing	2	2	00:04	48	4	timeout	62	3	00:04
Airbag	5	5	00:09	62	7	timeout	484	5	39:46
Odometer	3	3	00:07	6	3	timeout	13	3	00:08
FFU Star	16	16	00:47	110	41	timeout	4,768	289	timeout
FFU ECU	19	19	00:29	149	7	timeout	2	2	timeout
ASR ch1	2	2	01:50	31	2	timeout	9,607	20	timeout
ASR Reduced	3	3	57:57	27	3	timeout	34,972	7	timeout

Table 2. Experimental results for the algorithm variants 4), 3) and 2).

states completely is a significant increase in computation time. Algorithm variant 3), which corresponds to the approximating DSPM algorithm, experiences a timeout for all models, whereas CTBS in both variants 1) and 2) is able to analyze some models without timeout. This points to a performance advantage of CTBS over DPSM.

More generally, the advantage of CTBS over DPSM in terms of its performance on practical models can be argued as follows. First, the CTBS can return preliminary results. In particular, a causal trace set can be constructed when using a BFS state space exploration even when the computation aborts for instance due to a timeout. The analysis of the model `FFU Star` guarantees that all causal traces up to depth 37 when the timeout occurs are found, and that traces that were found up to that depth are actually causal. The same holds true for the analysis of model `FFU ECU` at depth 31, for the model `ASR ch1` at depth 37 and for the model `ASR Reduced` at depth 39. Notice that since we return causal trace sets in these situations, the results are complete and no causal traces up to the analysis depth that was reached are missing. Second, causal traces are minimal and, as a consequence, found early during the analysis. For several models the last causal trace was detected in less than 1 second, as can be gleaned from the data in column $Time_{Last}$. The remainder of the computation time is spent by the algorithm to ensure that the causal trace set contains all causal traces. For instance, for the `Airbag` model, the computation of this guarantee takes another 7 minutes and 9 seconds. For the models `FFU ECU` and `FFU Star` the checking of this guarantee timeouts after 2h. However, based on our knowledge of this model we can state that all causal traces were found by the time the timeout occurred.

The CTBS algorithm based on a BFS (variant 1) performs differently compared to the DFS based version (variant 2) in terms of runtime and causal results. For the `Airbag` model, variant 1) requires 7:09 which is substantially less than 39:46 for variant 2). A DFS based algorithm first searches the depth and checks many orderings of non causal traces before shorter causal traces are found. The CTBS algorithm based on DFS scales worse than based on BFS and this result goes in line with previous results in [LL13a]. Variant 2), however, found more causal classes than variant 1), for the `FFU Star`, `ASR ch1` and `ASR reduced` model. We had a closer look at the fault trees and detected that several causal traces returned by variant 2) are actually not causal. As expected, the

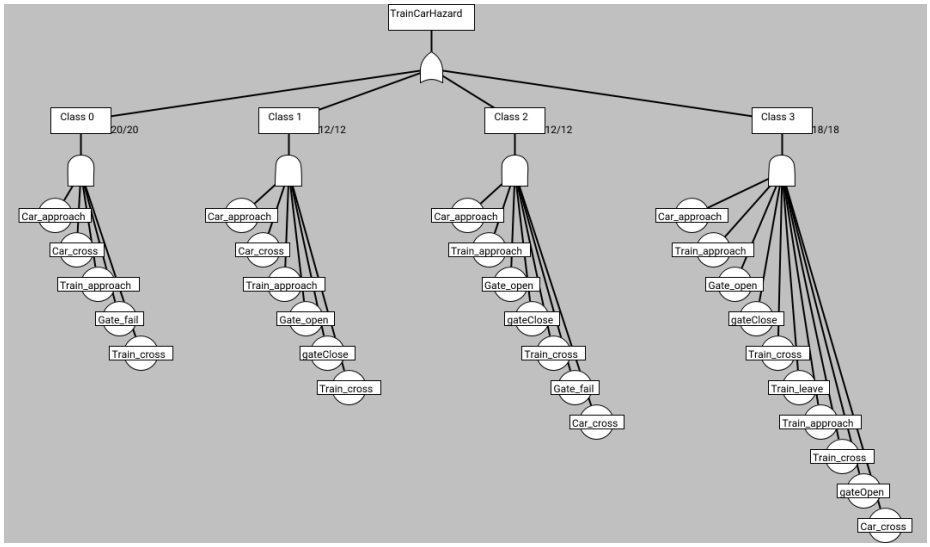


Fig. 2. Fault Tree for Railroad Crossing using CTBS variant 1)

DFS cannot ensure that a causal trace is causal until the algorithm terminates. We conclude that variant 2) is not sound when the analysis is aborted before completion, for instance due to a timeout.

Qualitative Result Interpretation. We refer to the railroad crossing example to illustrate some qualitative differences between variant 1) and variant 4), the algorithm defining the baseline in terms of quantitative performance. In the railroad crossing model, a train can approach the crossing (Train_approach), enter the crossing (Train_cross) and finally leave the crossing. Whenever a train is approaching, the gate should close (gateClose) and will open when the train has left the crossing. Additionally, it is possible that the gate fails (Gate_fail). The car approaches the crossing (Car_approach) and enters the crossing (Car_cross) when the gate is open, and finally leaves the crossing. The fault tree computed for the railroad crossing example by variant 4) is missing the causality classes labeled `Class 2` and `Class 3`, which were on the other hand computed by variant 1), see the fault tree in Figure 2. The events of `Class 0` are a subset of the events in `Class 2`, but the action Gate_fail happens after the action Train_cross. The events of `Class 1` are contained in `Class 3` but the action Car_approach happens after the action gateClose. When Gate_fail happens before Car_approach, the gate stays open. When Gate_fail happens after Car_approach then the gate falsely opens itself. From `Class 1` we can infer that an accident happens because the car is not leaving the crossing. From `Class 3` computed by CTBS in variant 1) we conclude that an accident can additionally happen because the train is not leaving the crossing. This exemplifies that CTBS in variant 1) can compute more information regarding the cause of an event in the system than the baseline

algorithm variant 4). This is due to the completeness of variant 1) as opposed to the incompleteness of variant 4).

5 Conclusion

In this paper we have addressed the complete computation of causal trace sets in Causality Checking. The complete computation of the causal trace sets is essential in the analysis of safety-critical systems in order to ensure that all causal factors will be identified by the analysis. The CTBS algorithm that we propose addresses the problem of a complete and sound construction of traces that belong to this set when the state space traversal encounters multiple duplicate states during the search. We contrast the CTBS algorithm with the DSPM algorithm, which is the current basis for implementations of Causality Checking. The variant of DSPM originally implemented in QuantUM only performs an incomplete under-approximative handling of the construction of traces when encountering duplicate states. In contrast, CTBS handles the encounter of duplicate states properly and completely and manages to establish complete causal sets. In particular, CTBS outperforms a variant of DSPM modified to accomplish a naive correction of the under-approximation.

Further research addresses the application of the algorithmic scheme underlying CTBS to other applications where all executions of a state space need to be explored exhaustively. Furthermore, we will investigate the computation of causal trace sets using symbolic state space exploration techniques.

References

AL10. Husain Aljazzar and Stefan Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. Software Eng.*, 36(1):37–60, 2010.

BHK+15. Adrian Beer, Stephan Heidinger, Uwe Kühne, Florian Leitner-Fischer, and Stefan Leue. Symbolic causality checking using bounded model checking. In *SPIN*, volume 9232 of *Lecture Notes in Computer Science*, pages 203–221. Springer, 2015.

BK+08. Christel Baier, Joost-Pieter Katoen, et al. *Principles of Model Checking*. MIT Press, 2008.

dJR10. Marc de Jonge and Theo C. Ruys. The spinja model checker. In *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer, 2010.

ES12. Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications*. Academic Press, 2012.

Fre60. Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.

Hal15. Joseph Y. Halpern. A modification of the halpern-pearl definition of causality. In *IJCAI*, pages 3022–3033. AAAI Press, 2015.

HK06. Henri Hansen and Antti Kervinen. Minimal counterexamples in o(n log n) memory and o(n^2) time. In *ACSD*, pages 133–142. IEEE Computer Society, 2006.

HKD09. Tingting Han, Joost-Pieter Katoen, and Berteun Damman. Counterexample generation in probabilistic model checking. *IEEE Trans. Software Eng.*, 35(2):241–257, 2009.

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

HP05. J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Phil. of Science*, 2005.

KL18. Martin Kölbl and Stefan Leue. Automated functional safety analysis of automated driving systems. In *FMICS*, volume 11119 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2018.

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

LL11. Florian Leitner-Fischer and Stefan Leue. Quantum: Quantitative safety analysis of UML models. In *QAPL*, volume 57 of *EPTCS*, pages 16–30, 2011.

LL13a. 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.

LL13b. Florian Leitner-Fischer and Stefan Leue. Probabilistic fault tree synthesis using causality computation. *IJCCBS*, 4(2):119–143, 2013.

LL14. Florian Leitner-Fischer and Stefan Leue. Spincause: a tool for causality checking. In *SPIN*, pages 117–120. ACM, 2014.

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

SB05. Viktor Schuppan and Armin Biere. Shortest counterexamples for symbolic model checking of LTL with past. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2005.

VGRH02. W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault Tree Handbook, 2002.