Technical Report soft-12-02, 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 towards 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 proposed in [10] that has been extended to reasoning about computational models in [13]. 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 [4]. 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 [17] and testing, on the one hand, and manual fault tree analysis [18] or failure mode and effect analysis [12] 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, 2] 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 the 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.

In this paper we leverage the benefit gained from analyzing large number of counterexamples for a given property. We are thus able to identify commonalities among them and might hence be able to gain insight into what combination of events may have caused 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 counterexamples 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 [16, 10].

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. The resulting causality relationships can be displayed as a fault tree.

While our approach towards an adoption of the counterfactual reasoning model that we use as well as a mapping of the causalities to fault trees has been presented in a precursory paper [13], this paper focuses on a refinement of our causality model and an integration of the causality computation into standard state space search as used by explicit-state model checkers. 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. Our on-the-fly algorithm for causality computation and how it is integrated in statespace exploration algorithms is presented in Section 3. How the computed causality relationships are represented by fault trees is explained in Section 4. In Section 5 we demonstrate the causality checking approach using two case studies. Related work is discussed throughout the paper and in Section 6. We conclude in Section 7.

2 Causality Reasoning in System Models

2.1 Causality Reasoning

Our goal is to identify the events that cause a system to reach a certain undesired state. Such a state could, for instance, represent a hazard or a potential unsafe state of the system. We use an explicit state model checker [11] to check whether there are system executions that lead to such an undesired state. In the philosophy of science, a commonly adopted notion of causality is that of *counterfactual* reasoning and the related *alternative world* semantics of Lewis [16, 5]. In software and systems engineering, the counterfactual argument is frequently used as the foundation for identifying faults in program debugging [19, 9]. 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 representing the alternative worlds.

In our precursory paper [13] we present an adaption of the *structural equation model (SEM)* by Halpern and Pearl [10] to the causality analysis of model

checking counterexamples. We use the adapted SEM in order to automatically compute fault trees out of sets of counterexamples. We refer to execution traces that lead to some effect as "bad" exection traces, all other traces are "good". The set of bad traces typically consists of the set of all counterexamples obtained during model checking. The underlying SEM extends the counterfactual reasoning approach by Lewis and encompasses the notion of causes being logical combinations of events as well as a distinction of relevant and irrelevant causes. In our precursory work, we extended the SEM by an event order logic that allows for considering the order of the events in causality computation. This extensions allows the application of this adapted causality model in the context of concurrent system models.

In [13] we have defined an event oder logic in order to describe logical constraints on event sequences. An ordered causal formula in the adapted SEM is an order constrained boolean conjunction ψ of boolean variables representing the occurrence of events. The variable associated with an event is true in case that event has occurred. ψ describes a logical condition that requires the events referred to inside ψ to occur, and to occur in a certain order.

[7] 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, computing causal relationships between variables is NP-complete. As a consequence, we compute an overapproximation of the precise causality. Instead of identifying single events that cause the effect, we compute the causal events together with the events that are part of what is referred to as the causal process. The causal process comprises all variables that mediate between the causal events and the effect that ψ is causing. Those variables are not root causes, but they contribute to propagating the cause through the system until the final effect is being triggered. For an event order logic formula representing an execution trace, the set of variables V, representing the occurrence of events, is partitioned into sets Z and W. The events represented by the variables in Z are the events we want to show for that they are part of the causal process.

For each trace there exists a function $val(V)$ and $order(V)$. $val(V)$ specifies the valuation of the variables in V. If a event occurs on the execution trace, $val_1(V)$ sets the value of the variable representing the event to true, respectively to false if it does not occur on the trace. $order_1(V)$ specifies the order of the events on the execution trace.

An event order logic formula ψ is considered a cause for an event represented by the event order logic formula φ, if the following conditions AC1, AC2(1), AC2(2) and AC3 are satisfied:

- AC1: Both ψ and φ are true in an execution trace given the valuation $val_1(V)$ and the order $order_1(V)$.
- AC2: There exists an execution trace with valuation $val_2(V)$ and an order $order_2(V)$ such that:
  1. Changing the values of the variables in Z and W from val_1 to val_2 and the order of the variables in Z and W from $order_1$ to $order_2$ changes φ from true to false.

2. Setting the values of the variables in W from val_1 to val_2 and the order of the variables in W from $order_1$ to $order_2$ should have no effect on φ as long as the values of the variables in Z are kept at the values defined by val_1, and the order as defined by $order_1$. If changing the values of the variables in W from val_1 to val_2 and the order of the variables in W from $order_1$ to $order_2$ does have an effect on φ, we take the variables in $Q \subset W$ for which $val_1(Q) \neq val_2(Q)$ and $order_1(Q) \neq order_2(Q)$ and add their negated values to Z.

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

If a formula ψ meets conditions AC1 through AC3, the occurrence of the events included in ψ is causal for φ. However, condition AC2 does not imply that the order of the occurring events is causal. A subset $Y \subset Z$ has an influence on the causality for φ if OC1 holds for that subset:

OC1: Let $Y \subseteq Z$. Changing the order $order_1(Y)$ of the variables in Y to an arbitrary order $order_2(Y)$, while keeping the variables in $Z \setminus Y$ at $order_1$, changes φ from true to false.

2.2 Running Example

In the running example of a railroad crossing system that we use in the paper, a train can approach the crossing (Ta) and 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) but 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.

Suppose we want to check whether the execution trace "Ta,Ca,Gf,Cc,Tc" is causal. AC1 is fulfilled, since "Ta,Ca,Gf,Cc,Tc" is an actual execution trace of the model, and both the train and the car are in the crossing at the same time.

In order to show that AC2(1) is fulfilled, we choose the valuation $val_2(V)$ defined by the execution "Ta,Ca,Gf,Cc". $val_2(V)$ sets the value of the variable representing the event Tc to false and the value of φ changes from true to false since now only the car is in the crossing, but not the train.

We now need to test AC2(2) for the execution trace "Ta,Ca,Gf,Cc,Tc". We change the values of the variables in W to the valuation $val_2(V)$ and $order_2(V)$ specified by the execution trace "Ta,Ca,Gf,Cc,Cl,Tc". In the execution "Ta,Ca,Gf,Cc,Cl,Tc" the car leaves the crossing before the train enters the crossing and no crash occurs. Consequently, the value of φ is changed from true to false. The set of the variables in $Q \subset W$ for which $val_1(Q) \neq val_2(Q)$ and $order_1(Q) \neq order_2(Q)$ consists of the variable representing the event "Cl". Hence we add the negation of that event "¬Cl" to the set Z and get the execution trace "Ta,Ca,Gf,Cc,¬Cl,Tc".

In our model there does not exist an execution trace that is shorter than the execution trace "Ta,Ca,Gf,Cc,¬Cl,Tc" and leads to an accident, hence no subset of Z satisfies conditions AC1 and AC2 and consequently AC3 is fulfilled.

Finally we need to check the order of events with OC1. 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 will be caused.

3 On-The-Fly Causality Checking

3.1 Preliminaries

For the off-line causality computation described in [13] all good and bad execution traces need to be computed and stored on disk prior to the causality checking. In this section we outline an algorithm for causality computation along with a suitable data structure that can be integrated in an on-the-fly depth-first search or breadth-first search algorithm. It is then no longer necessary to store all good and bad execution traces before performing the causality computation.

In order to compute causality relationships, it is still 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. Note that we want to analyze reachability properties and hence only need to consider finite execution fragments [2]. Bad executions are all those executions where a property violation is detected. There are two categories of good executions that need to be distinguished. The first and obvious one is when the search reaches a state that has no successor states, without finding a property violation on the trace to this state. The second category of good executions is what we call *so-far good executions*. The so-far good executions are executions where there has not yet been a property violation so far. In other words, these executions could be prefixes of either bad or good executions.

Note that the tests defined in Section 2 require to find or establish sub- or super-set relationships between the different system executions. In the following we also use the terms sub-execution and super-executions to refer to sub- or super-set relationships between different system executions. For AC2(1) we need to find good sub-executions of the execution trace under test, since we remove events from Z by changing $val_1(V)$ to $val_2(V)$. To test AC2(2) we need to find good super-executions where events in W have been added to the execution trace. And in AC(3) we have to show that there is no sub-execution that fulfills AC1, AC2(1) and AC2(2).

Inferring causality in this model based setting can be seen as the problem of finding sub- or super-executions and determining whether the found executions are bad or good. We use this observation to devise an algorithm and a corresponding data-structure called sub-set graph (Section 3.2) for on-the-fly causality checking with breadth-first (Section 3.3) and depth-first search (Section 3.4).

3.2 Sub-Set Graph Data-Structure

For the storage of the execution traces we have devised a data-structure called sub-set 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 sub-set graph. The sub-set 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 sub-set graph for the railroad crossing example. The execution traces on adjoining levels are connected by edges indicating sub-set relationships between the respective execution traces. For simplicity reasons the edges between executions on the same level are not displayed in the figure. The nodes representing the

Fig. 1. Sub-set-graph of the railroad crossing example.

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

1. 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 the trace "Ca,Ta,Gc,Tc,Tl" of our example. Similarly to the black traces the green traces can not be causal because they are good traces. The green traces can be prefixes of either bad or good execution traces.
2. 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 statespace exploration. They are considered to be causal. As an example consider the trace "Ta,Ca,Gf,Cc,Tc" of our example.
3. 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.
4. 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 sub-set of the trace "Ca,Ta,Gc,Tc,Tl,Go,Ta,Gf,Cc,Tc", hence the trace does not fulfill the minimality constraint.

The causality checking that we propose is embedded into both of the standard statespace exploration algorithms, namely depth-first and breadth-first search. The pseudo-code of the subset-graph is shown in Figure 1 Whenever a bad, a good or a so-far good execution is found by the search algorithm it is added to that level of the subset graph that corresponds to its length. 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 sub-set relationships of the execution already in the sub-set graph hold also for the newly added execution. For instance in our example all sub-set 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 sub-set 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 sub-set 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 sub-set 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 for sub-set relationships with all other traces that are colored red, since it might be the case that a shorter red trace exists on a lower level and was not continued because it ended in the bad

state. 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.

All executions that are colored red are candidates for being causal and fulfill AC1, AC2(2) and AC3. For each execution trace that is colored red AC1 is fulfilled by definition, since the events that occur on the execution trace are being considered as ψ and for the execution trace being red the effect φ also has to occur. AC2(1) is fulfilled by each execution trace that is colored red, because all sub execution traces on the level below represent execution traces where events from the red execution trace have been removed, that corresponds to setting X to val_2. Since it is required by the definition that for an execution trace being colored red all sub executions are colored green, the outcome of φ can be changed by changing variables in X, which is sufficient to satisfy AC2(1). If breadth-first search is used, and therefore the shortest paths are added first, AC3 is fulfilled for all execution traces that are colored red. This is due to the fact that by definition an execution trace is only colored red if all its sub-sets are colored green, which means there is no sub-set where the effect occurs. If depth-first search is used, AC3 holds as soon as the statespace exploration is finished and we can be sure that no shorter traces can be found.

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

```
class SubSetGraph
{

HashMap <Integer, Trace> traces; // stores all traces
List of List of Integers levelToIDs; // stores ids on different level
List of Integers = redTraces; // stores red traces on different levels
boolean searchNotCompleted = true;

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

Once the state space search is completed we have to perform the tests for AC2(2) and OC1 for all red execution traces. For the AC2(2) test all direct black successors (level+1) need to be considered. If the test AC2(2) is fulfilled the execution is causal. Note that in our context the AC2(2) test is needed to detect whether the nonoccurence of an event is causal. To test AC2(2) we need to store all traces that are colored black, that otherwise would not have to be stored. Hence we have added a runtime switch in our implementation that allows to turn the AC2(2) test off to save memory at the cost of not being able to take the possible causality of the nonoccurence of an event into account. After the AC2(2) test the OC1 test is performed for the execution trace. Due to the

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
   int level = length(t);
   for each t' in getTracesByLevel(level){
          Trace tempTrace;

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

  /* Find all direct subsets */

  if(subSetSearchNeeded ) {
      boolean subsetFound = false;
      for each t' in getTracesByLevel(level-1) {
          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;
          }
        }
```

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

```
      if(searchMode == DFS)
      {
        for each t' in getTracesByLevel(level+1) {
          Trace tempTrace;

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

            if(t'.color == red && t.isBad()){
                 t'.color = orange;
                remove(redTraces, t');
            }else if (t'.color == green){
                 t'.color = black;
            }
            subsetFound = true;
          }
        }
      }
  }
  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);
      }
    }
  }
}
```

Algorithm 4 Algorithm sketch: isSubset(Trace t', Trace t)

```
function boolean isSubset(Trace t', Trace t)
{ // is trace t' a subset of trace t
        if(length(t') > length(t))
        {
                return false;
        }

        List events = t'.getEvents();

        FOR ALL e in events
        {
                if(!eventExistsInTrace(t, e))
                {
                        return false;
                }
        }

        return true;
}
```

Algorithm 5 Algorithm sketch: eventExistsInTrace(Trace t, event e)

```
function boolean eventExistsInTrace(Trace t, event e)
{
        List events = t.getEvents();

        FOR ALL e' in events
        {
                if(e = e')
                {
                        return true;
                }
        }

        return false;
}
```

structure of the sub-set 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 the order of this pair is marked as having no influence on causality.

Algorithm 6 Algorithm sketch of the checkAC22() method.

```
function checkAC22()
{
  FOR ALL Trace t in redTraces
  {
    FOR ALL Trace t_1 in
        getBlackSuperSets(t)
    {
        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 7 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 redTraces
    {
      if (t.getID() != t_1.getID() && length(t) == length(t_1))
      {
        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 j = i+1 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);
                  redTraces.remove(t);
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Algorithm 8 Algorithm sketch: isEqual(Trace t', Trace t)

```
function boolean isEqual(Trace t', Trace t)
{
        if(length(t') != length(t))
        {
                return false;
        }
        else
        {
                List events = t'.getEvents();

                FOR ALL e in events
                {
                        if(!eventExistsInTrace(t, e))
                        {
                                return false;
                        }
                }
        }
        return true;
}
```

Algorithm 9 Algorithm sketch: isOrderedEqual(Trace t, Trace t')

```
function boolean isOrderedEqual(Trace t, Trace t')
{
        if(length(t) != length(t'))
        {
                return false;
        }

        events_t = t.getEvents();
        events_t' = t'.getEvents();

        FOR i = 0 to size(events_t)
        {
                if(events_t.get(i) != events_t'.get(i)))
                {
                        return false;
                }
        }
        return true;
}
```

3.3 Causality Checking with Breadth-First Search (BFS)

The pseudo-code of the adapted breadth-first search algorithm is given in the Algorithm 10 figure. When using breadth-first search, the execution trace leading

to a state can be generated by iterating through the predecessor states. Whenever a bad, a good or a so-far good execution is found it is added to the sub-set graph. If BFS encounters a state that is already in the statespace and hence all successors of this states have been already explored, the successors are not explored a second time. Since BFS explores the statespace 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 reaching the state is longer than the already known execution trace. If the new execution trace has the same length, the events on the trace have a different order as in the already known execution trace. Hence the new execution trace needs to be added to the sub-set graph because we need the trace for the OC1 test.

3.4 Causality Checking with Depth-First Search (DFS)

We adapted the depth-first search algorithm typically used in checking reachability properties in explicit-state model checking to add an execution trace to the sub-set graph data structure whenever either a bad state is reached, a "so far"-good execution trace had been found, or a good execution trace is encountered. 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 state that is already in the statespace and hence all successors of this states have already been explored, the successors are not explored a second time. It is possible that this new trace to the state is shorter or has a different event order as the already known execution traces that contain the same state. Hence we store this new execution trace on a match list in the sub-set graph and generate all execution traces starting from this new state with the new trace as a prefix.

4 Representing Causality with Fault Trees

4.1 Fault Trees and Fault Tree Analysis

In order to achieve acceptance in practice we need to use a notation to represent the causality information that we compute that is widely known in systems engineering. Fault trees (FTs) [18] are being used extensively in industrial systens engineering practice, in particular in fault prediction and analysis, to illustrate graphically under which conditions systems can fail, or have failed. Fault trees document graphically which combination of events can cause a system hazard. We hence use fault trees to represent the computed causality relationships.

In our context, we need the following elements of the FT notation:

1. Basic event: represents an atomic event.
2. *AND* gate: represents a failure if all of its input elements fail.
3. *OR* gate: represents a failure if at least one of its input elements fails.
4. Priority-AND (*PAND*): represents a failure if all of its input elements fail in the specified order. The required input failure order is usually read from left to right.

Algorithm 10 Algorithm sketch of the adapted breadth first search algorithm.

```
Queue D = {}, Statespace 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*/
    }
}
```

Algorithm 11 Algorithm sketch of the extended depth first search algorithm.

```
SubSetGraph G = {}
Statespace 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
}
```

5. Intermediate Event: failure events that are caused by their child nodes. A top level event (TLE) is a special case of an intermediate event, representing the system hazard.

The graphical representation of these elements can be found in Fig. 2. The *AND*, *OR* and *PAND* gates are used to express that their top events are caused by their input events. For an in-depth discussion of fault trees we refer the reader to [18].

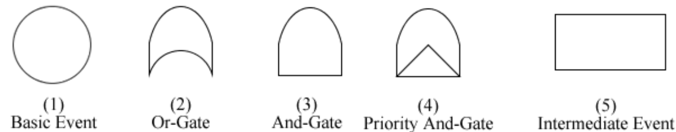

Fig. 2. Fault Tree Elements

4.2 Mapping of Causal Execution Traces to Fault Trees

All execution traces that are colored red are part of the fault tree and have to be included in the fault tree representation. The fault trees generated by our approach all have a normal form, that is they start with an *intermediate* gate representing the top level event, that is connected to an *OR* gate. The execution traces that are colored red are added to the fault tree as follows: execution traces with a length of one and hence consisting of only one basic event are represented by the respective basic event. An execution trace with length greater than one that has no subset of labels marked as ordered is represented by an *AND* gate. This *AND* gate connects the basic events belonging to that execution trace. If a (subset of a) execution trace is marked as ordered it is represented by a *PAND* gate that connects the basic events in addition with an *Order Condition* connected to the *PAND* gate constraining the order of the elements. Subsequently, an intermediate event is added as parent node for each *AND* gate and *PAND* gate. The resulting intermediate events are then connected by an *OR* gate that leads to the top event, representing the hazard.

5 Case Studies

5.1 Experiment Setup

In order to evaluate the proposed approach, we have implemented our causality checking algorithms within the SpinJa toolset [6], a Java re-implementation of the explicit state model checker Spin [11]. Our SpinCause tool[1] computes the

[1] Our SpinCause tool and the promela models of the case studies are available at http://www.inf.uni-konstanz.de/soft/tools/spincause.

causality relationships for a Promela model and a given LTL reachability property. The following experiments were performed on a PC with an Intel Xeon Processor (3.60 Ghz) and 144 GBs RAM.

5.2 Railway Crossing

Figure 3 shows the fault tree of the railway crossing example that was generated by our tool. For better readability we have omitted the order constraints of the *PAND*-gates. There are two combinations of events, that cause a crash on the railway crossing:

- First, if a train (Ta) and a car (Ca) are approaching and the gate fails (Gf), this results in a hazardous situation where both the car (Cc) and train (Tc) are on the crossing at the same time.
- 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 before the gate is closing (Cl) and the train is crossing (Tc), this also corresponds to a hazardous situation.

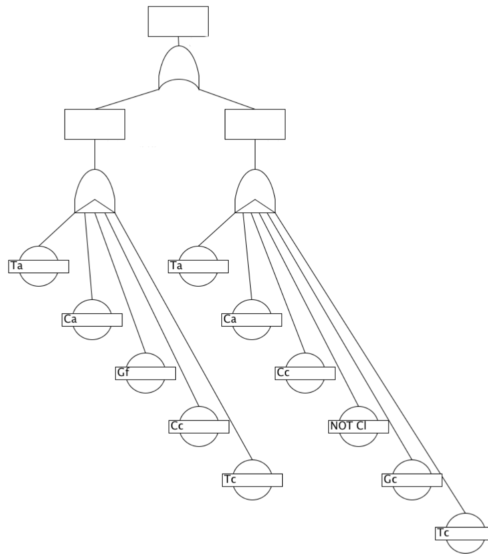

Fig. 3. Fault Tree of the railway crossing running example.

Figure 4 shows the experimental results for the railway model. The columns Run. MC and Mem. MC show the runtime and memory consumption if only model checking is done. The runtime and memory that is needed to perform model checking and causality checking when the AC2(2) test is disabled is shown by the columns Run. CC1 and Mem. CC1, respectively Run. CC2 and Mem. CC2 when the AC2(2) test is enabled.

	Run. MC	Mem. MC	Run. CC 1	Mem. CC 1	Run. CC 2	Mem. CC 2
DFS	0.01 sec.	16.40 MB	0.29 sec	20.38 MB	0.31	21.68 MB
BFS	0.01 sec.	16.24 MB	0.12 sec.	16.70 MB	0.13 sec	17.45 MB

Fig. 4. Experiment results for the railway crossing case study.

5.3 Airbag Control Unit

The industrial size model of an airbag system that we use in this case study is taken from [1]. 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 lives in crash situations, they may cause fatal accidents if they are inadvertently deployed. It is therefore a pivotal safety requirement that an airbag is never deployed if there is no crash situation. We are interested in generating the fault tree for an inadvertent ignition of the airbag. Notice that the Promela model was automatically synthesized from a higher-level design model in UML, by our QuantUM tool [15].

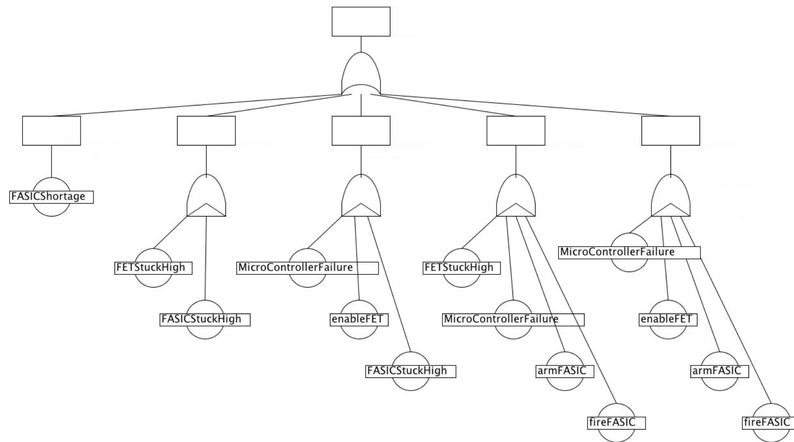

Fig. 5. Fault tree of the airbag system

Figure 5 shows the fault tree generated by our tool. For better readability we have omitted the order constraints on the *PAND* gates. 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*. It is also easy to see that the combination of the basic events *FETStuckHigh*

and *FASICStuckHigh* only lead to an inadvertent deployment of the airbag if the basic event *FETStuckHigh* occurs prior to the basic event *FASICStuckHigh*. The case study shows that the fault tree is a compact and concise visualization of the counterexample which allows for an easy identification of the basic events that cause the inadvertent deployment of the airbag. If the order of the events is important, this can be seen in the fault tree by the *PAND* gate. Figure 6 shows the experimental results for the airbag model. The columns Run. MC and Mem. MC show the runtime and memory consumption if only model checking is done. The runtime and memory that is needed to perform model checking and causality checking when the AC2(2) test is disabled is shown by the columns Run. CC1 and Mem. CC1, respectively Run. CC2 and Mem. CC2 when the AC2(2) test is enabled.

	Run. MC	Mem. MC	Run. CC 1	Mem. CC 1	Run. CC 2	Mem. CC 2
DFS	0.98 sec.	25.08 MB	338.17 sec.	15,711.20 MB	597.57 sec.	27,687.50 MB
BFS	0.96 sec.	25.74 MB	148.52 sec.	1,597.54 MB	195.05 sec.	3,523.04 MB

Fig. 6. Experiment results for the airbag case study.

Figure 7 shows the memory and run time consumption for model checking and causality checking if the approach presented in [13] is used, where the execution traces are stored on disk during model checking and the causality checking tool reads that file once the model checking is done and performs the causality checking off-line.

	Run. MC	Mem. MC	Run. Causality	Mem. Causality
DFS	871.14 sec.	1,478.34 MB	945.68 sec.	28,563.47 MB
BFS	486.01 sec.	1,331.29 MB	512.3 sec.	13,860.10 MB

Fig. 7. Experiment results for off-line causality checking of the airbag case study.

5.4 Discussion

Both case studies show the similar trends:

- If no causality checking is done, DFS and BFS have approximately the same runtime and memory consumption.
- When performing causality checking, BFS outperforms DFS in terms of both runtime and memory consumption. This is due to the fact that if BFS is used, the algorithm can rely on assumptions regarding the length of the execution traces, namely that BFS always finds shortest counterexamples. If a trace is added by BFS, the algorithn can rely on that all traces that will be added after the trace have equal or greater length. Consequently all execution traces

that are added by BFS and colored red immediately fulfill the minimality condition AC3. When using DFS we can not make any assumptions on the length of the paths. It is possible that bad traces are colored red but later in the search a shorter bad trace is found. If the shorter bad trace is a sub-execution of the longer bad traces the minimality condition is violated for the longer traces. The longer traces would not have been considered to be causal if BFS would have been used. This effect becomes especially obvious in terms of memory consumption if we take the non-occurrence of events into account. If we want to check AC2(2) we have to store all good execution traces that are super-sets of bad execution traces. When using BFS we know for each trace all possible sub-traces and hence only have to store the black traces. If we use DFS we do not know all sub-traces, because we always might find a shorter trace later in the search, and hence have to store all good execution traces.

The online approach described in this paper outperforms the off-line approach from [13] both in terms of runtime and memory consumption. This is because we do not have to store all execution traces on disk during the model checking and read them back into memory for causality checking.

6 Related Work

Work documented in [3] uses the Halpern and Pearl approach to explain counterexamples in functional CTL model checking by determining causality. However, this approach considers only functional counterexamples that consist of single execution sequences. In [8] a formal framework for reasoning about contract violations is presented. In order to derive causality the notion of precedence established by Lamport clocks [14] is used. While this captures a partial order of the observed contract violations there is no evidence whether this partial order has an impact on causality or not. Work described in [9] establishes causality based on Lewis counterfactual reasoning by computing distance metrics between execution traces. The delta between the counterexample and the most similar good execution is identified as causal for the bad behavior. 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 statespace exploration algorithms and which works on-the-fly.

7 Conclusions

We have discussed how causality relationships can be established in system executions. Furthermore we have proposed an approach for causality computation that works on-the-fly and can be integrated into explicit statespace model checking algorithms. Additionally we have shown how the computed causality relationships can be represented with fault trees. 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, which is due to the fact that assumptions on the length of the execution traces can be made if breadth-first search is used. In addition, the experimental evaluation also shows that the on-the-fly approach we presented here outperforms our precursory off-line approach.

In future work we plan to 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.

References

1. H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In *Proc. of QEST 2009*. IEEE Computer Society, 2009.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
3. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *Proceedings of CAV 2009*, LNCS. Springer, 2009.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking (3rd ed.).* The MIT Press, 2001.
5. J. Collins, editor. *Causation and Counterfactuals.* MIT Press, 2004.
6. M. de Jonge and T. Ruys. The spinja model checker. In *Model Checking Software*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer, 2010.
7. T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. *Artificial Intelligence*, 2002.
8. G. Gössler, D. L. Métayer, and J.-B. Raclet. Causality analysis in contract violation. In *Runtime Verification*, volume 6418 of *LNCS*, pages 270–284. Springer Verlag, 2010.
9. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3), 2006.
10. J. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 2005.
11. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addision–Wesley, 2003.
12. International Electrotechnical Commission. Analysis Techniques for System Reliability - Procedure for Failure Mode and Effects analysis (FMEA), IEC 60812, 1991.
13. 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*, LNCS. Springer, 2011.
14. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.
15. 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)*, volume 57 of *EPTCS*, pages 16–30, 2011.
16. D. Lewis. *Counterfactuals.* Wiley-Blackwell, 2001.

17. D. Parnas, A. van Schouwen, and S. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, 1990.
18. U.S. Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.
19. A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.