Towards Symbolic Causality Checking using SAT-Solving

Adrian Beer$^1$, Uwe Kühne$^2$, Florian Leitner-Fischer$^1$, Stefan Leue$^1$

$^1$University of Konstanz, $^2$Airbus - Defence and Space

Abstract: With the increasing size and complexity of modern safety critical embedded systems, the need for automated analysis methods is growing as well. Causality Checking is an automated technique for formal causality analysis of system models. In this paper we report on work in progress towards an Symbolic Causality Checking approach. The proposed approach is based on bounded model checking using SAT solving which is known to be efficient for large and complex system models.

1 Introduction

The size and complexity of modern software-driven and safety critical systems is increasing at a high rate. In this situation, classical manual safety analysis techniques like reviewing, fault tree analysis [VGRH02] and failure mode and effect analysis [In91] can only be applied to very limited parts of the architecture of a system. Furthermore, these techniques are more suitable for analyzing faults in hardware systems rather than in software driven embedded systems. The demand for automated methods and tools supporting the safety analysis of the architecture of software-driven safety-critical systems is growing.

In previous work, an algorithmic, automated safety-analysis technique called causality checking was proposed [LFL13a]. Causality checking is based on model checking. In model checking, the model of the system is given in a model checker specific input language. The property is typically given in some temporal logic. The model checker verifies whether the model acts within the given specifications by systematically generating the state space of the model. If the model does not fulfill the specification, an error trace leading from the initial state of the model to the property violation is generated. One trace only represents one execution of the system. In order to understand all possibilities of how an error can occur in a system, all possible error traces have to be generated and inspected. Manually locating reasons for property violations using these traces is problematic since they are often long, and typically large numbers of them can be generated in complex systems. Causality checking is an algorithmic, automated analysis technique working on system traces which supports explaining why a system violates a property. It uses an adaption of the notion of actual causality proposed by Halpern and Pearl [HP05]. The result of the causality checking algorithm is a combination of events that are causal for an error to happen. The event combinations are represented by formulae in Event Order Logic (EOL) [LFL13c], which can be fully translated into LTL, as is shown in [BLFL14]. The EOL formulae produced by causality checking represent the causal events in a more compact way than counterexamples since they only contain the events and the relation between those events that are considered to be causal for a property violation. It was shown
that the explicit-state causality checking approach is efficient for system models for which state-of-the-art explicit model checking is efficient as well [LFL13a].

Although the explicit-state causality checking method was shown to be efficient for small to medium sized models, for system models that cannot be efficiently processed by explicit state model checkers the causality computation is also not efficient. In this paper we propose a new causality checking approach based on Bounded Model Checking (BMC) [BCCZ99]. BMC can efficiently find errors in very large systems where explicit model checking runs out of resources. One drawback of BMC is that it is not a complete technique since it cannot prove the absence of errors in a system beyond a predefined bound on the length of the considered execution traces. For the proposed symbolic causality checking method this means that completeness for the computed causalities can only be guaranteed for system runs up to the given bound. In explicit causality checking all traces through a system have to be generated in order to gain insight into the causal events. The symbolic causality checking approach presented in this paper uses the underlying SAT-solver of the bounded model checker in order to generate the causal event combinations in an iterative manner. This means that only those error traces are generated that give new insight into the system. Traces that do not give new information are automatically excluded from the bounded model checking algorithm by constraining the SAT-solver with the already known information. With this technique a large number of error traces can be ruled out that would need to be considered in the explicit approach, which contributes to the efficiency of the symbolic approach. We implemented this approach as an addition to the NuSMV2 model checker [CCG+02].

In [LFL11a, LFL11b] we presented a tool based approach called QuantUM that allows for specification of dependability characteristics and requirements directly within a system or software architecture modeled in UML [uml10] or SysML [Sys10]. The system models are automatically translated into the input language of different model checkers, for instance the model checker SPIN [Hol03]. Afterwards, the integrated explicit causality checker calculates the causal events for a property violation and displays the results in terms of dynamic Fault Trees [VGRH02]. In [LFL13b] a combination of the explicit causality checking and the probability computation was shown where probabilities for the causality classes can be calculated. The probabilities can be tagged to the Fault Trees. The integration of causality checking into the QuantUM tool chain enables the applicability of causality computations in an model-based engineering environment. The integration of the symbolic causality checking presented in this paper can be done in a similar way.

The remainder of the paper is structured as follows. In Section 2 we will present the foundations of our work which includes bounded model checking and the notion of causality. Section 3 is devoted to the new symbolic approach to causality computation. In Section 4 we evaluate the symbolic approach in comparison to the explicit causality checking. Related work will be discussed in Section 5 before we conclude in Section 6.

## 2 Preliminaries

### 2.1 Running Example

We will illustrate the formal framework that we present in this paper using the running example of a simple railroad crossing system. In this system, a train can approach the
crossing \((T_a)\), enter the crossing \((T_c)\), and finally leave the crossing \((T_l)\). Whenever a train is approaching, the gate shall close \((G_c)\) and will open again when the train has left the crossing \((G_o)\). It might also be the case that the gate fails \((G_f)\). The car approaches the crossing \((C_a)\) and crosses the crossing if the gate is open \((C_c)\) and finally leaves the crossing \((C_l)\). We are interested in finding those events that are causal for the hazard that the car and the train are in the crossing at the same time.

2.2 System Model

The system model that we use in this paper is that of a transition system \([BK08]\):

**Definition 1** (Transition System). A transition system \(M\) is a tuple \((S, A, \rightarrow, I, AP, L)\) where \(S\) is a finite set of states, \(A\) is a finite set of actions/events, \(\rightarrow \subseteq S \times A \times S\) is a transition relation, \(I \subseteq S\) is the set of initial states, \(AP\) is the set of atomic propositions, and \(L : S \rightarrow 2^{AP}\) is a labeling function.

**Definition 2** (Execution Trace). An execution trace \(\pi\) in \(M\) is defined as an alternating sequence of states \(s \in S\) and actions \(a \in A\) ending with a state. \(\pi = s_0 \alpha_1 s_1 \alpha_2 s_2 \ldots \alpha_n s_n, \text{s.t. } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i < n.\)

An execution sequence which ends in a property violation is called an error trace or a counterexample. In the railroad crossing example, \(s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5\) is a counterexample, because the train and the car are inside the crossing at the same time.

2.3 Linear Temporal Logic

We use the standard syntax and semantics of the Linear Temporal Logic (LTL) as introduced by Pnueli \([Pnu77]\). The operators \(\bigcirc, \square, \Diamond\) and \(U\) are used to express temporal behavior, such as “in the next state sth. happens”(\(\bigcirc\)), “eventually sth. happens”(\(\square\)) or “sth. is always true”(\(\Diamond\)). The \(U\)-operator denotes the case that “\(\varphi_1\) has to be true until \(\varphi_2\) holds”.

There are two non-disjoint classes of LTL properties, safety and liveness properties. Safety properties can be violated by a finite prefix of an infinite path, while liveness properties can only be violated by an infinite path. For now, causality checking has only been defined for safety properties.

The property that we want to express in the railroad crossing is that the train and the car shall never be in the crossing at the same time: \(\square \neg (T_c \land C_c)\).

2.4 Event Order Logic

The Event Order Logic \([LFL13]\) (EOL) can be fully translated into LTL as was shown in \([BLFL14]\). EOL captures the occurrence and order of events on a trace through a transition system.

**Definition 3** (Syntax of the Event Order Logic). Simple event order logic formulae are defined over the set \(A\) of event variables:

\[
\phi ::= a \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \neg \phi
\]
where $a \in A$ and $\phi_1$ and $\phi_2$ are simple EOL formulae. Complex EOL formulae are formed according to the following grammar:

$$\psi ::= \phi | \psi_1 \land \psi_1 | \psi_1 \lor \psi_2 | \psi_1 \land \psi_2 | \psi_1 \land (\phi_1 \land \psi_1) | \psi_1 \land \psi_1 \land (\phi_1 \land \psi_1)$$

where $\phi$ is a simple EOL formula and $\psi_1$ and $\psi_2$ are complex EOL formulae.

We define that a transition system $M$ satisfies the EOL formula $\psi$, written as $M \models_e \psi$, iff $\exists \pi \in M. \pi \models_e \psi$. The informal semantics of the operators can be given as follows.

- $\psi_1 \land \psi_2$: $\psi_1$ has to happen before $\psi_2$.
- $\psi_1 \land \psi_1$: $\psi_1$ has to happen at some point and afterwards $\phi$ holds forever.
- $\phi \land \psi_1$: $\phi$ has to hold until $\psi_1$ holds.
- $\psi_1 \land \phi \land \psi_2$: $\psi_1$ has to happen before $\psi_2$, and between $\psi_1$ and $\psi_2$, $\phi$ has to hold all the time.

For example, the formula $Gc \land Tc$ states that the gate has to close before the train enters the crossing. The full formal semantics definition for EOL is given in [LFL13c].

### 2.5 Causality Reasoning

Our goal is to identify the events that cause a system to reach a property violating state. Therefore, it is necessary to formally define what “cause” means in our context. We will use the same definition of causality that was proposed in [KLFL11] as an extension of the structural equation model by Halpern and Pearl [HP05]. In particular this extension accounts for considering the order of events in a trace to be causal. For example, an event $a$ may always occur before an event $b$ for an error to happen, but if $b$ occurs first and $a$ afterwards the error disappears. In this case, $a$ occurring before $b$ is considered to be causal for the error to happen.

**Definition 4 (Cause for a property violation [HP05, LFL13a])**. Let $\pi, \pi'$ and $\pi''$ be paths in a transition system $M$. The set of event variables is partitioned into sets $Z$ and $W$. The variables in $Z$ are involved in the causal process for a property violation while the variables in $W$ are not. The valuations of the variables along a path $\pi$ are represented by $\text{val}_z(\pi)$ and $\text{val}_w(\pi)$, respectively. $\psi_\lambda$ denotes the rewriting of an EOL formula $\psi$ where the ordering operator $\land$ is replaced by the normal EOL operator $\land$, all other EOL operators are left unchanged. An EOL formula $\psi$ consisting of event variables $X \subseteq Z$ is considered to be a cause for an effect represented by the violation of an LTL property $\phi$, if the following conditions hold:

- **AC 1**: There exists an execution $\pi$ for which both $\pi \models_e \psi$ and $\pi \not\models_e \phi_1 \phi$.
- **AC 2.1**: $\exists \pi'$. s.t. $\pi' \not\models_e \psi \land (\text{val}_z(\pi) \not\models_e \text{val}_w(\pi)) \land \text{val}_w(\pi')$ and $\pi' \models_e \phi_1 \phi$. In other words, there exists an execution $\pi'$ where the order and occurrence of events is different from execution $\pi$ and $\phi$ is not violated on $\pi'$.
- **AC 2.2**: $\exists \pi''$ with $\pi'' \not\models_e \psi \land (\text{val}_z(\pi) = \text{val}_w(\pi) \land \text{val}_w(\pi'))$ and $\pi'' \models_e \phi$ it holds that $\pi'' \not\models_e \phi$ for all subsets of $W$. In words, for all executions where the events in $X$ have the value defined by $\text{val}_z(\pi)$ and the order defined by $\psi$, the value and order of an arbitrary subset of events on $W$ has no effect on the violation of $\phi$.
- **AC 3**: The set of variables $X \subseteq Z$ is minimal: no subset of $X$ satisfies conditions AC 1 and AC 2.
• **OC 1:** The order of events represented by the EOL formula $\psi$ is not causal if the following holds: $\pi \Vdash_e \psi$ and $\pi' \not\Vdash_e \psi$ and $\pi' \not\Vdash_e \psi \land$

The EOL formula $Gf \land (\langle Ta \land (Ca \land Cc) \rangle \langle \land \neg Cl \langle \land Tc \rangle)$ is a cause for the occurrence of the hazard in the railroad crossing example since it fulfills all of the above defined conditions (AC 1-3, OC 1).

2.6 Bounded Model Checking

The basic idea of Bounded Model Checking (BMC) [BCCZ99] is to find error traces, also called counterexamples, in executions of a given system model where the length of the traces that are analyzed are bounded by some integer $k$. If no counterexample is found for some bound $k$, it is increased until either a counterexample is found or some pre-defined upper bound is reached. The BMC problem is efficiently reduced to a propositional satisfiability problem, and can be solved using propositional SAT solvers. Modern SAT solvers can handle satisfiability problems in the order of $10^9$ variables.

Given a transition system $M$, an LTL formula $f$ and a bound $k$, the propositional formula of the system is represented by $[[M,f]]_k$. Let $s_0, ..., s_k$ be a finite sequence of states on a path $\pi$. Each $s_i$ represents a state at time step $i$ and consists of an assignment of truth values to the set of state variables. The formula $[[M,f]]_k$ encodes a constraint on $s_0, ..., s_k$ such that $[[M,f]]_k$ is satisfiable iff $\pi$ is a witness for $f$. The propositional formula $[[M,f]]_k$ is generated by unrolling the transition relation of the original model $M$ and integrate the LTL property in every step $s_i$ of the unrolling. The generated formula $[[M,f]]_k$ of the whole system is passed into a propositional SAT solver. The SAT solver tries to solve $[[M,f]]_k$. If a solution exists, this solution is considered to be a witness to the encoded LTL property.

3 Symbolic Causality Checking

3.1 Event Order Normal Form

In order to efficiently store the event orderings and occurrences in the symbolic causality algorithm it is necessary to use a normal form. This normal form is called event order normal form (EONF) [BLFL14]. EONF permits the unordered and- ($\land$) and or-operator (\lor) only to appear in a formula if they are not sub formulas in any ordered operator and only and-operators (\land) if they are sub formulas of the between operators $\langle \land \rangle$ and $\langle \land \rangle$. For instance, the EOL formula $Ta \land Gc \land Tc$ can be expressed in EONF as $\psi_{EONF} = (Ta \land Gc) \land (Gc \land Tc) \land (Ta \land Tc)$.

3.2 EOL Matrix

For the symbolic causality computation with bound $k$ we focus on sequence of events $\pi_e = e_1 e_2 e_3 \ldots e_k$ derived from paths of type $\pi = s_0 \overset{e_1}{\rightarrow} s_1 \overset{e_2}{\rightarrow} s_2 \ldots$. We use a matrix in order to represent the ordering and occurrence of events along a trace. This matrix is called EOL matrix.

**Definition 5** (EOL matrix). Let $E = \{e_1, e_2, e_3, \ldots, e_k\}$ an event set and $\pi_e = e_1 e_2 e_3 \ldots e_k$ the corresponding sequence. The function $o$ is defined for entries where $i \neq j$ and the func-
$$d(i,j) = \begin{cases} \{ \text{TRUE} \} & \text{if } e_i \wedge e_j \\ \phi & \text{if } e_i \wedge \phi \wedge e_j \\ \emptyset & \text{otherwise} \end{cases}$$

The EOL matrix $$M_E$$ is created as follows:

$$M_E = \begin{pmatrix}
    d(e_1) & o(e_1,e_2) & \ldots & o(e_1,e_k) \\
    o(e_2,e_1) & d(e_2) & \ldots & o(e_2,e_k) \\
    \vdots & \vdots & \ddots & \vdots \\
    o(e_k,e_1) & o(e_k,e_2) & \ldots & d(e_k)
\end{pmatrix}$$

where the generated entries in the matrix are sets of events or the constant set $${\{\text{TRUE}\}}$$. The empty set $$\emptyset$$ indicates that no relation for the corresponding event configuration was found.

The special case $$e \wedge \phi$$ is not considered in function $$d$$ because this will never occur when analyzing safety properties.

**Definition 6** (Union of EOL Matrices). Let $$M_E, M_{E_1}, M_{E_2}$$ be EOL Matrices with the same dimensions. The EOL matrix $$M_E$$ is the union of $$M_{E_1}$$ and $$M_{E_2}$$ according to the following rule:

$$M_{E(i,j)} = M_{E_1(i,j)} \cup M_{E_2(i,j)} \quad (1)$$

for every entry $$(i,j)$$ in the matrices.

The union of two EOL matrices represents the component-wise disjunction of two matrices. The EOL matrix $$M_E$$ for an example event sequence in the railroad crossing $$\pi = \text{Ca Cc Gf}$$ and a refinement EOL Matrix $$M'_E = M_E \cup M_{\pi'}$$ using the sequence $$\pi' = \text{Gf Ca Cc}$$ is created as follows:

$$e_1 = \text{Ca} \quad e_2 = \text{Cc} \quad e_3 = \text{Gf}$$

$$M_E = \begin{pmatrix}
    \emptyset & \{\text{TRUE}\} & \{\text{TRUE}\} \\
    \emptyset & \emptyset & \{\text{TRUE}\} \\
    \emptyset & \emptyset & \emptyset
\end{pmatrix} \quad M'_E = \begin{pmatrix}
    \emptyset & \{\text{TRUE}\} & \{\text{TRUE}\} \\
    \emptyset & \emptyset & \emptyset \\
    \{\text{TRUE}\} & \{\text{TRUE}\} & \emptyset
\end{pmatrix} \quad (2)$$

The information stored in a EOL matrix can be translated back into an EOL formula in EONF. As was shown in [BLFL14] every EOL formula can be translated into an equivalent LTL formula. This translated LTL formula is then translated further into propositional logic [BCCZ99].

### 3.3 The Algorithm

In Figure [1] the informal iteration schema of the proposed algorithm is presented. The inputs to the algorithm are the model $$M$$, the property $$\phi$$ to check and an upper bound $$k_{\text{max}}$$ for the maximum length of individual counterexamples (CX).

1. the algorithm starts at level $$k = 0$$.
2. If no CX is found the bound is increased until the next CX is found.
3. The CX is transformed into a EOL formula in EONF and saved in a EOL Matrix.
4. The new EOL matrix is used to refine a matching, already found EOL matrix (see Definition 6) or to set up a new class of causes [LFL13c].
5. In the next iteration the event orderings in the matrices are translated into propositional logic formulas and inserted into the SAT solver in order to strengthen the constraints and, thus, find possible new orderings or new event combinations.
6. The disjunction over all EOL matrices represents the set of all computed causes of errors.

3.4 Soundness and Completeness

The following informal thoughts can be proven similar to [LFL13c]: From the definition we conclude that each CX that is found satisfies AC 1. By structural induction over the generation of the EOL matrices one can prove that new CX are always the shortest new CX that can be found and there does not exist a shorter sequence of events that lead into the property violation under the given constraints. Therefore, the minimality constraint AC 3 is fulfilled by the EOL matrices. AC 2.1 is fulfilled by each CX, since, if the last event on the CX is removed there exists a path containing a sub set of the events which does not end in a hazard state. The only problem left to solve is the AC 2.2 condition.

Event Non-Occurrence Detection. According to the AC 2.2 test the occurrence of events that are not considered as causal must not prevent the effect from happening. In other words, the non-occurrence of an event can be causal for a property violation. Therefore, we have to search such events and include their non-occurrence in the EOL formulas. In Figure 2 an example is presented which explains this procedure for an EOL formula $\psi = Ca \land Cc \land Ta \land Gc \land Tc$. Trace 1 is the minimal trace ending in a property violation. Trace 2 is non-minimal and also ends in a property violation with the events $Ca, Cc, Ta, Gc, Gf, Tc$. In trace 3 a new event $Cl$ appears between $Cc$ and $Tc$ and no property violation is detected. This means that the appearance of the event has prevented the property violation. In order to transform this appearance into a cause for the hazard, the occurrence is negated and introduced into the EOL formula $\psi = \ldots Cc \land \neg Cl \land \neg Ta \ldots$. The new clause
Figure 2: Three example traces for the EOL-formula $\psi = Ca \land Cc \land Ta \land Gc \land Tc$. Trace 1 is the minimal trace. While trace 2 (non-minimal) ends in a property violation, trace 3 does not.

states that “if between ‘the car is on the crossing’ and ‘the train is approaching the crossing’, ‘the car does NOT leave the crossing’, the hazard does happen”. In other words: The non-occurrence of $Cl$ is causal for the property violation.

For every level $k$ a second pass of the algorithm needs to be done in order to find the non-occurrences. The input parameters are altered compared to the first pass. Now the algorithm searches for paths that fulfill the property $\phi$ and the constraints from the EOL matrices. With this inputs the algorithm finds traces that fulfill the EOL formula and the property which must be due to an event which prevents the property violation form happening.

4 Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Run time (sec.)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MC</td>
<td>CC1</td>
</tr>
<tr>
<td>Railroad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>explicit</td>
<td>0.01</td>
<td>0.12</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.01</td>
<td>0.16</td>
</tr>
<tr>
<td>Airbag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>explicit</td>
<td>0.96</td>
<td>148.52</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.02</td>
<td>4.81</td>
</tr>
</tbody>
</table>

Table 1: Experimental results comparing the explicit state approach in the best case according to [LFL13a] to the symbolic approach for the railway crossing and airbag case studies.

In order to evaluate the proposed approach, we have implemented the symbolic causality checking algorithm within the symbolic model checker NuSMV2 [CCG+02]. Our CauSeMV extension of NuSMV2 computes the causality relationships for a given NuSMV model and an LTL property. The NuSMV models used in the experiments were generated manually. In practical usage scenarios the NuSMV models may be automatically derived from higher-level design models, as for example with the QuantUM tool [LFL11b].

As first case study we consider the railroad crossing example from Section 2.1. The second case study is the model of an industrial Airbag Control Unit taken from [AFG+09]. All experiments were performed on a PC with an Intel Xeon Processor (3.60 Ghz) and 144GBs of RAM. We compare our results with the results for the explicit state causality checking approach presented in [LFL13a], which were performed on the same computer.

Table 1 presents a comparison of the computational resources needed to perform the explicit and the symbolic causality checking approaches. Run. MC and Mem. MC show the runtime and memory consumption for model checking only. Run. CC1 and Mem. CC1 show the runtime and memory needed to perform causality checking without the AC2(2) condition and Run. CC2 and Mem. CC2 with the AC2(2) test enabled.

The results illustrate that for the comparatively small railroad crossing example the explicit state causality checking finishes faster and uses less memory than in the symbolic approach. For the larger airbag model the symbolic approach outperforms the explicit
address both in terms of time and memory.

5 Related Work

In [BBDC+09, GMR10, GCKS06] a notion of causality was used to explain the violations of properties in different scenarios. While [BBDC+09, GCKS06] use symbolic techniques for the counterexample computation, they focus on explaining the causal relationships for a single counterexample and thus only give partial information on the causes for a property violation. All of the aforementioned techniques rely on the generation of the counterexamples prior to the causality analysis while our approach computes the necessary counterexamples on-the-fly. In [BV03] and [BCT07], a symbolic approach to generate Fault Trees [VGRH02] is presented. In this approach all single component failures have to be known in advance while in our approach these failures are computed as a result of the algorithm. The ordering and the non-occurrence of events cannot be detected in this approach as being causal for a property violation.

6 Conclusion and Future Work

We have discussed how causal relationships in a system can be established using symbolic system and cause representations together with bounded model checking. The symbolic causality checking approach was evaluated on two case studies and compared to the explicit state causality checking approach. The symbolic causality checking can be used in an integrated tool chain, called QuantUM, in order to fully automatize the verification of systems modeled in UML / SysML and further automatically generate Fault Trees containing the causes for a system failure.

References


