Causality Checking of Safety-Critical Software and Systems

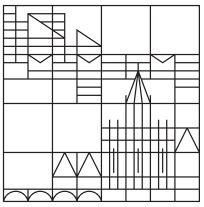
Dissertation submitted for the degree of
Doctor of Engineering (Dr.-Ing.)

Presented by

Florian Leitner-Fischer

at the

Universität Konstanz



Faculty of Sciences
Department of Computer and Information Science

Date of the oral examination: 6th of March 2015

First referee:	Prof. Dr. Stefan Leue, University of Konstanz, Konstanz, Germany
Second referee:	Prof. Dr. Alberto Lluch-Lafuente, Technical University of Denmark, Denmark

To my family

Acknowledgments

Silent gratitude isn't much use to anyone.
- G.B. Stern -

Writing a PhD thesis is a great adventure and like most adventures it is even more fun if there are great companions that are there to discuss and share ideas, have a chat or grab ab coffee, and most important to provide support and encouragement. I had several of such companions and I want to thank all of them.

Firstly and foremost I want to express my gratitude to my advisor Prof. Dr. Stefan Leue for his outstanding guidance and mentoring, his valuable advice and comments, but also for giving me the freedom to find my own way and to work on my own ideas.

I would like to thank Prof. Dr. Alberto Lluch-Lafuente for acting as a second referee and for his valuable and insightful comments. In addition I thank the chair of my PhD committee Prof. Dr. Marc Scholl.

I want to thank our industry partners at Airbus Defence & Space, BMW, Daimler and TRW Automotive for giving me the chance to see my research in action and to apply it to real world problems and for the valuable feedback.

I thank the current and former members of the software and systems engineering group at the University of Konstanz. Particularly, I want to thank my colleagues Dr. Husain Aljazzar, Dr. Matthias Kuntz and Dr. Wei Wei for the great time we had together in my early days in the group and for everything that I learned from them. I thank Dr. Nafees Ur Rehman for the coffee breaks and for always having a story or a joke to tell. Special thanks also to Adrian Beer for being a great colleague, for the great time we had working on QuantUM, traveling to conferences and workshops and most of all for being a great friend.

I am indebted to my family for the support and encouragement during the last years, but especially for the support in the last few months. Without you renovating a house, while starting a new job and preparing a PhD thesis defense, with a newborn child at home would have been impossible.

My special gratitude goes to my wife, Viktoria, for her continuous love and encouragement, for listening to me when I couldn't stop talking about my research ideas, but most of all for the greatest gift in the world, our son John.
John, I also want to thank you, your smile is the most motivating, cheerful and adorable thing that I have ever seen[1].

[1] And I want to also thank you for giving me enough sleep while I had to prepare my PhD defense.

Abstract

The complexity of modern safety-critical systems is steadily increasing due to the amount of functionality that is implemented in those systems. 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, a formal verification technique, provides an automated algorithmic analysis of system models and is able to check whether a formalized requirement is satisfied by the system. If the requirement is violated the model checker provides error traces, called counterexamples, which serve as witnesses for the requirement violation. While the counterexamples can be used as a debugging aid, they do not provide any obvious insight into which events are causal for the requirement violation. In order to derive the causal events from a set of counterexamples, the events of all error traces need to be manually inspected, which is an error prone and time consuming task.

We propose a method complementing model checking that is called causality checking. Causality checking algorithmically computes the causal events for the violation of a requirement that is formalized as a non-reachability property. The notion of causality that we use is based on the actual cause definition by Halpern and Pearl, which in turn is based on the counterfactual argument by Lewis. Causality checking is able to compute the causal events for a property violation and also takes the order in which the events appear into account as a causal factor. Furthermore, causality checking identifies the causal non-occurrence of events. We propose a causality checking algorithm and show how it can be integrated into the state-space exploration algorithms used in qualitative model checking.

The hardware of an embedded system may fail due to deterioration and affect the software that is running on this hardware and thereby cause a requirement violation. In industry negative exponentially distributed rates are used in order to specify the occurrence probability of such a hardware failure in a given time frame. In order to support a probabilistic analysis of the system we extend the causality checking approach to be applicable to probabilistic system models. The proposed probabilistic causality checking approach enables us to compute the probabilities of certain combinations of events that cause a property violation. A pure probabilistic causality checking method, as will become clear in this thesis, entails a high performance penalty for the necessary probabilistic counterexample computation. We will show how this bottleneck can be mitigated by a combination of qualitative and probabilistic causality checking.

While a full state-space exploration of the case studies that we analyze is possible, we discuss the implications of an incomplete state-space exploration of the analysis model with respect to soundness and completeness of the causality checking result and present strategies that can be applied in order to increase the scalability of the causality checking approach.

Furthermore, we discuss how the causality checking method can be integrated

into the QuantUM framework, a method that allows for the generation of the analysis models from higher-level architectural description languages such as the unified modeling language (UML) or the systems modeling language (SysML). For the result representation we propose a temporal logic called event order logic which allows for a concise and formal representation of the causal events and the causal orderings. Furthermore, we show how formulas in event order logic, that have been computed by the causality checker, can be represented by fault trees, a method used in industry to reason about causal relationships between property violations and events. The mapping to fault trees facilitates the interpretation of the causality checking results and is a representation that is already known and used in industry.

We demonstrate the applicability and usefulness of the causality checking approach on several case studies from industry and academia and discuss how causality checking can be used in an industrial safety engineering process.

Zusammenfassung

Die Komplexität moderner sicherheitskritischer Systeme steigt, aufgrund der Menge an Funktionalität, welche in diesen Systemen implementiert ist, stetig an. Um in der Lage zu sein, die Korrektheit und Sicherheit dieser System in einer umfassenden Art und Weise zu beurteilen, bedarf es an automatisierten oder zumindest rechnergestützten Techniken. Model Checking, eine formale Verifikationstechnik, stellt eine automatische, algorithmische Analyse von Systemmodellen zur Verfügung und ermöglicht es zu prüfen, ob eine formale Anforderung vom System erfüllt wird. Falls die Anforderung verletzt wird, liefert der Model Checker eine fehlerhafte Ausführungsfolge, ein sogenanntes Gegenbeispiel, welche als Nachweis für die Anforderungsverletzung dient. Obwohl Gegenbeispiele die Fehlersuche unterstützten, geben sie keinen Aufschluss darüber, welche Ereignisse kausal für die Anforderungsverletzung sind. Um die kausalen Ereignisse aus einer Menge von Gegenbeispielen herzuleiten, müssen die Ereignisse aller fehlerhaften Aufführungsfolgen untersucht werden. Dies ist eine fehleranfällige und zeitaufwendige Aufgabe.

Wir schlagen eine Methode namens Causality Checking vor, welche das Model Checking ergänzt. Causality Checking berechnet algorithmisch die kausalen Ereignisse für die Verletzung einer als Nicht-Erreichbarkeits-Eigenschaft formalisierten Anforderung. Der Kausalitätsbegriff den wir benutzen basiert auf der "actual cause"-Definition von Halpern und Pearl, welche wiederum auf dem kontrafaktischen Argument von Lewis basiert. Mittels Causality Checking ist es möglich die kausalen Ereignisse für eine Eigenschaftsverletzung zu berechnen und zusätzlich die Ordnung der Ereignisse als kausalen Faktor zu berücksichtigen. Außerdem erkennt Causality Checking das kausale nicht-auftreten von Ereignissen. Wir schlagen einen Causality Checking Algorithmus vor und zeigen, wie sich dieser in die Algorithmen zur Zustandsraumexploration, welche für das qualitative Model Checking benutzt werden, integrieren lässt.

Die Hardware eines eingebetteten Systems kann durch Alterung ausfallen und die Software, welche auf der Hardware ausgeführt wird, beinträchtigen und so eine Anforderungsverletzung verursachen. In der Industrie werden Raten mit negativer Exponentialverteilungen benutzt um die Auftretenswahrscheinlichkeit eines solchen Hardwarefehlers innerhalb eines bestimmten Zeitrahmen zu spezifizieren. Um eine probabilistische Analyse des Systems zu ermöglichen, erweitern wir den Causality Checking Ansatz, um ihn für probabilistische Systemmodelle anwendbar zu machen. Der vorgeschlagene probabilistische Causality Checking Ansatz ermöglicht es die Wahrscheinlichkeiten bestimmter Kombinationen von Ereignissen, welche kausal für die Eigenschaftsverletzung sind zu berechnen. In dieser Arbeit zeigt sich, dass eine reine probabilistische Causality Checking Methode, aufgrund der notwendigen Berechnung der probabilistischen Gegenbeispielen zu hohen Leistungseinbussen führt. Wir zeigen wie, dieser Flaschenhals durch eine Kombination aus qualitativem und probabilistischem Causality Checking umgangen werden kann.

Obwohl eine vollständige Exploration des Zustandsraum für die analysierten Fallstudien möglich ist, diskutieren wir welche Auswirkungen eine unvollständige Exploration des Zustandsraums des Analyse Models auf die Korrektheit und die Vollständigkeit des Causality Checking Ergebnis hat und präsentieren Strategien die angewendet werden können um die Skalierbarkeit des Causality Checking Ansatzes zu erhöhen.

Zudem diskutieren wir, wie die Causality Checking Methode in den QuantUM Ansatz, einer Methode zur Generierung von Analysemodellen aus Architekturbeschreibungssprachen auf höherer Ebene, wie die Unified Modeling Language (UML) oder die Systems Modeling Language (SysML), integriert werden kann. Zur Ergebnisrepräsentation schlagen wir eine temporale Logik namens Event Order Logic vor, welche eine präzise und formale Repräsentation der kausalen Ereignisse und der kausalen Ordnungen ermöglicht. Zusätzlich zeigen wir, wie Formeln in der Event Order Logic, welche vom Causality Checker berechnet wurden, als Fehlerbäume dargestellt werden können. Fehlerbäume sind eine in der Industrie verwendete Methode, um Kausalitätsbeziehungen zwischen Eigenschaftsverletzungen und Ereignissen darzustellen. Die Abbildung auf Fehlerbäume vereinfacht die Interpretation des Causality Checking Ergebnisses und ist eine Darstellung, die in der Industrie bereits verwendet wird.

Wir demonstrieren die Verwendbarkeit und den Nutzen des Causality Checking Ansatzes an einigen akademischen und industriellen Fallstudien und diskutieren, wie Causality Checking in einem industriellen Sicherheitsanalyseprozess eingebunden werden kann.

Contents

Chapter 1

Introduction

Contents

1.1 Motivation

Nowadays software intensive embedded systems control cars, aircraft, trains, nuclear power plants and many other systems. A failure of such a system may lead to a catastrophic crash, the loss of life, severe injuries, great environmental, or financial damage. Such systems are also called *safety-critical systems*.

Ensuring the safe and correct functionality of safety-critical systems is of paramount importance and is governed by development standards for safety-critical systems, such as IEC 61508 [54] applicable to electrical systems, DO-178C/ED-12C [91] for software in airborne systems, CENELEC EN 50128 [27] for railway systems or the ISO 26262 [55] for automotive systems.

The complexity of modern safety-critical systems is steadily increasing due to the amount of functionality that is implemented in those systems. With the increasing complexity of safety-critical systems, the need for methods which support engineers to asses the safe and correct functionality of the systems becomes evident. Due to the size and complexity of the systems traditional techniques for safety analysis and fault localization like reviews [51], testing, manual fault tree analysis [97] or failure mode and effect analysis [53], 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 [8, 31] 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 a property violating state. 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 and represents merely one possible execution of the system. In order to deduce all causal event combinations for a property violation using model checking, one has to manually analyze and compare all possible counterexamples generated by the model checker, which is a time-intensive and tedious task. In addition, the number of the possible counterexamples generated by the model checker is usually very large, even for small models.

The goal of this thesis is to develop an approach that allows for the identification of the events that are causal for the violation of a system property. We propose a method called *causality checking* which aims at providing insight into why a property was violated during model checking. Causality checking uses an adaption of the *actual cause* definition by Halpern and Pearl [46] in order to algorithmically compute the causal events for a property violation. The actual cause definition by Halpern and Pearl [46] is based on the counterfactual reasoning argument and the related alternative world semantics of Lewis [32, 81]. The 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. Halpern and Pearl extend the Lewis counterfactual model to what they refer to as *structural equation model* (SEM) [46]. It encompasses the notion of *actual causes* being logical combinations of events as well as a distinction of relevant and irrelevant causes. In the SEM events are represented by variable values and the minimal number of causal variable valuation combinations is determined. In order to do so the counterfactual test is extended by contingencies. Contingencies can be viewed as possible alternative worlds, where a variable value is changed. A variable X is causal if there exists a contingency, that is a variable valuation for other variables, that makes X counterfactual. We adapt the actual cause definition from [46] so that it can be applied to the system models used for model checking. The systems that we aim to analyze are concurrent systems and, consequently, the order in which the events occur also needs to be considered as a causal factor for the property violation. This is necessary since one execution trace might entail a race condition and lead to a property violation while another execution trace consisting of the same events might not violate the property. Consequently, we extend the adapted actual cause definition in order to take the ordering of events into account as a causal factor. We propose an algorithm that computes the events causing a property violation, together with the order in which the events have to occur to be causal. Furthermore, we show how this algorithm can be integrated into the state-space exploration algorithms used for model checking. When analyzing a model of, for instance, a railroad crossing the causal events for a crash identified by the causality checker are, for example, a train and a car are approaching the railroad crossing, the gate securing the railroad crossing fails to close, and both the car and the train enter the crossing. In addition, the causality checker will, for instance,

show that it is not causal whether the train or the car is approaching the crossing first.

While causality checking gives an answer to the question "which events, in what order, cause a property violation?" it does not give any information on the frequency or likelihood of the events causing the property violation. For software systems we can assume that a software once it was verified will not deteriorate. This assumption no longer holds for software in an embedded system, since the hardware in which the software is embedded might deteriorate and exhibit certain failure behavior that impacts the software, such as for example bit flips in the memory that is used by the software. In industry negative exponentially distributed rates, also called failure rates, are used to estimate the occurrence probability of such hardware failures in a given time frame [95, 96]. In previous work [2] it was shown that probabilistic model checking [8] can be used to compute the probability of a property violation and probabilistic counterexamples [5, 47] can be used for the debugging of the system. In the case of probabilistic model checking the debugging of the system becomes even more difficult. While in qualitative model checking a single trace often provides valuable information for the debugging of the system, a single trace is most often not sufficient to form a probabilistic counterexample. Due to the fact that the violation of a probabilistic property with a probability-bound can hardly ever be traced back to a single error trace. In almost all cases a set of error traces is needed to provide an accumulated probability mass that violates the probability-bound of the specified probabilistic property. With an increasing number of error traces that are needed to form the probabilistic counterexample, an increasing number of different error traces need to be manually retraced and interpreted in order to gain insight into why the property was violated. We extend our causality checking approach to be applicable to probabilistic system models. The proposed *probabilistic causality checking* approach enables us to compute the probabilities of certain combinations of events that cause a property violation.

Two major concerns that have to be considered when developing a method for the analysis of systems are on one hand, how the analysis models needed for the analysis can be generated in a straightforward way and how the results of the analysis can be represented in a concise and easy to understand form. Thus we integrate our causality checking method into the QuantUM framework [72], a method that allows for the generation of the analysis models from higher-level architectural description languages such as the unified modeling language (UML) [86] or the systems modeling language (SysML) [52]. For the result representation we propose a temporal logic called *event order logic* which allows for a concise and formal representation of the causal events and the causal orderings. Furthermore, we show how formulas in event order logic, that have been computed by the causality checker, can be represented by fault trees [97], a method used in industry to reason about causal relationships between property violations and events. The mapping to fault trees facilitates the interpretation of the causality checking results and is a representation that is already known and used in industry.

1.2 Contributions

The contributions of this thesis can be summarized as follows:

- The event order logic, proposed in this thesis, is a temporal logic that allows to formally capture the occurrence and order of events and is used to represent the results of the causality checking method.
- We show how causal relationships can be inferred in system models based on an adapted version of the actual cause definition by Halpern and Pearl and how the order of the events can be taken into account as a causal factor.
- We propose a causality checking algorithm and show how it can be integrated into the state-space exploration algorithms used in qualitative model checking.
- Furthermore, we extend the causality checking method in order to be applicable to probabilistic system models.
- A pure probabilistic causality checking method, as will become clear in this thesis, entails a high performance penalty for the necessary probabilistic counterexample computation. We will show how this bottleneck can be mitigated by a combination of qualitative and probabilistic causality checking.
- We will demonstrate the applicability and usefulness of causality checking on several case studies from industry and academia and discuss how causality checking can be pushed beyond the limits of scalability.
- In order to make the results of causality checking easy to interpret for engineers, we propose a mapping of event order logic formulas to fault trees, a method used in industry to reason about the relationships between a property violation and the corresponding causal events.
- In addition, we discuss how causality checking can be used in an industrial setting, and how it can be integrated into the QuantUM method, a framework for the automated safety analysis of system and software architectures.

1.3 Outline

Chapter 2 presents the state of the art and related work.

Chapter 3 contains a brief introduction into the foundations on which causality checking is build upon.

Chapter 4 proposes the event order logic and discusses the relationship of event order logic with linear temporal logic and ω-automata.

Chapter 5 discusses how causal relationships can be inferred in formal system models.

Chapter 6 presents a causality checking algorithm that is integrated into the state-space exploration algorithms used for qualitative model checking.

Chapter 7 shows how causality checking can be applied to probabilistic system models.

Chapter 8 proposes a combination of qualitative and probabilistic causality checking which reduces the memory and runtime that is needed in order to perform probabilistic causality checking.

Chapter 9 discusses how causality checking behaves at the limits of scalability and proposes strategies to increase the scalability for practical application scenarios.

Chapter 10 shows how the event order logic formulas returned by the causality checker can be mapped to fault trees.

Chapter 11 discusses how causality checking can be used in an industrial setting and shows how causality checking can be integrated into the QuantUM method.

Chapter 12 concludes the thesis and suggests future work.

1.4 Own Publications

Parts of this thesis have been previously published in [13, 14, 16, 61, 62, 73, 74, 75, 76, 77, 78, 79, 80]. All of the aforementioned publications were co-authored and supervised by Stefan Leue. Our first results about probabilistic causality analysis were published in [61] and an extended version in [62]. In these papers we discuss how causal relationships can be inferred from a set of good- and bad-execution traces and present a first version of the event order logic. Matthias Kuntz provided initial ideas for the probabilistic analysis part of [61, 62]. Furthermore, we show how event order logic formulas can be mapped to fault trees. A refined version of the probabilistic causality checking algorithm was published in [78]. The work published in [61, 62, 78] is included in Chapter 4, Chapter 5, Chapter 7, and Chapter 10. The qualitative causality checking approach described in Chapter 6 was first proposed in [75] further refined in [74] and finally published in [76]. The combination of the qualitative causality checking approach presented in Chapter 8 was published in [77] and an extended version in [79]. In [16], which results from joint work with Adrian Beer, the syntax and semantics of the event order logic is refined and the relationship of event order logic and linear temporal logic is discussed, Adrian Beer contributed some initial ideas on the event order logic to linear temporal logic translation. Chapter 4 includes the work from [16]. In [13, 14] causality checking was applied to the industrial case study of an airport surveillance radar, we use this jointly authored case study by Beer et al. for experimental evaluation in Chapter 6, Chapter 7, Chapter 8, and Chapter 9. The software tool *SpinCause* which implements the algorithms proposed in this thesis was described in [80] and can be downloaded

from the URL `http://se.uni-konstanz.de/research1/tools/spincause/`. The contents of Chapter 9 have not been previously published. Chapter 10 contains unpublished work on the refinement of causality relationships, the relationship to minimal cut set analysis and a graphical representation of event order logic formula. Parts of the discussion in Chapter 11, on how causality checking and the QuantUM approach can be used within the context of the ISO 26262, have previously been published in [73].

Chapter 2

State of the Art and Related Work

Contents

2.1 Causality Reasoning in System Models

There are two important use cases for formal verification techniques, like model checking [31]. The first one is the verification of the correctness of the system, where the goal is to prove that there are no errors in the system. The second one is the debugging of the system. While in the first case it is sufficient, if the formal verification tool reports whether a formalized property is satisfied or not, additional information is needed in order to debug the system. For the debugging of the system it is necessary to have an intuition of why the system is violating the property. While it is sufficient to detect one error at a time when debugging a system, this is not sufficient if the goal is to present evidence that can be used to compile a safety case for a safety-critical system. A safety case [54] is a structured argument that demonstrates that the a system is acceptably safe. In order to do so all possible combination of events that can lead to a property violation have to be known and it has to be shown that these events are either controlled with some fault tolerance or safety mechanism or that their occurrence probability is low enough to be acceptable.

Traditionally when a property is violated, the formal verification tool will return error traces or counterexamples. By manually retracing the different steps of the counterexamples one can derive an intuition on why the property was violated in a particular case. Usually such a manual debugging process using counterexamples is an iterative and time consuming task. There are several possible approaches known in the literature that can be used to automatically explain a property violation. One possibility is to perform a static analysis of the system, like it is done by Ball et al. in [9]. Ball et al. propose a method that uses static analysis to localize errors in counterexamples and that can be applied to error traces of C programs. Another

possibility is to combine systematic debugging techniques and data mining as was done by Zeller in [104]. Third it is possible to use a dynamic analysis technique, such as model checking, and to compare the counterexamples generated by the model checker in order to derive an explanation for the property violation as it was done, among others, by Beer et al. in [17, 18] and Groce et al. in [45].

All of the aforementioned approaches have in common that they either implicitly or explicitly use some sort of causality reasoning in order to explain the cause for the property violation. The are several notions of causality that are used in the literature on error localization and debugging. Examples include the notion of precedence established by Lamport clocks [70], which while it captures a partial order of the observed events there is no evidence whether this partial order has an impact on causality or not. Petri nets [89] and event structures [102] are used in [85] to establish causality in system models. Another very commonly used notion of causality used in the analysis of system models is the counterfactual reasoning argument and the related alternative world semantics of Lewis [32, 81]. The naive interpretation of the Lewis counterfactual test, however, leads to a number of inadequate or even fallacious inferences of causes, in particular if causes are given by logical conditions on the combinations of multiple events. The problematic issues include common or hidden causes, the disjunction and conjunction of causal events, the non-occurrence of events, and the preemption of failure causes due to, e.g., repair mechanisms. A comprehensive discussion of these issues can be found in Section 5.1 and in the critical literature on counterfactual reasoning, e.g., [32]. The actual cause definition by Halpern and Pearl [46] is based on the counterfactual reasoning argument of Lewis and encompasses the notion of causes being logical combinations of events as well as a distinction of relevant and irrelevant causes and thus solves the shortcomings of the counterfactual reasoning argument. A shortcoming of the actual cause definition by Halpern and Pearl is that it does not take the order in which the events occur into account as a causal factor and, hence, is not directly applicable to concurrent systems.

In [44] a formal framework for reasoning about global contract violations is presented. In order to derive causality the notion of precedence established by Lamport clocks [70] is used. The proposed algorithm decides whether a prefix of a local trace is the cause for a global property violation or not, thus focusing on individual traces instead of a set of traces. The main objective of the causality definitions in [44] is to assign blame for a contract violation which entails that no alternative sequence is allowed to violate the contract or else it is not possible to assign the blame to one of the sequences. The approach from [44] is extended in [100] in order to support causality analysis of component-based real-time systems.

The approaches proposed in [45, 60, 104] are based on the counterfactual reasoning argument by Lewis. Due to the shortcomings of the naive counterfactual reasoning the approaches in [45, 60, 104] do not support complex logical relationships as causes. Furthermore, Zeller's approach [104] is mainly applicable to sequential software programs, and is not directly applicable to concurrent software and hardware systems. Work by Groce et al. published in [45] establishes causality based on

counterfactual reasoning by computing distance metrics between execution traces. The delta between the counterexample and the most similar good execution is identified as causal for the bad behavior. The main difference between the approach by Groce et al. and our approach is that Groce et al. aim at explaining causal relationships in a single counterexample whereas we aim at identifying all possible causal combinations of events that can lead to a property violation. Similarly to [45] in [60] an approach is proposed where the delta of a good and a bad execution trace is used to localize errors in the behavior model of a system.

Work documented in [17, 18] uses the Halpern and Pearl approach to explain counterexamples in computational tree logic model checking by determining causality. However, this approach considers only single counterexamples. Furthermore, it focuses on the causality of variable value-changes for the violation of computational tree logic sub-formulas. Consider the example of a railroad crossing where the computational tree logic formula representing the hazardous state where both the train and the car are on the crossing at the same time, consists of the two boolean variables train_on_crossing and car_on_crossing. Obviously, both variables changing to true is causal for a crash. The approach proposed in [17] will indicate the variable value-change of train_on_crossing and car_on_crossing from false to true as being causal. However, this obvious answer does not give any insight on why the train and the car are on the crossing at the same time. A notion for causality very similar to the actual cause definition by Halpern and Pearl is proposed in [21] for the explanation of data-flow based counterexamples of SCADE models [41]. Chockler et al. define in [29] a coverage measure for model checking based on the actual cause definition by Halpern and Pearl, which allows to assign a component of a system a quantitative measure of its relevance to the satisfaction of a property.

For all the above mentioned approaches it is necessary to compute the counterexamples prior to the causality analysis, which is then performed as a post-processing step. To the best of our knowledge we are not aware of any other causality checking algorithm that can be integrated with explicit state-space exploration algorithms and which works on-the-fly.

The approaches described in [24, 25, 42, 88] do not aim at identify the causal events for a property violation, but instead verify whether some manually specified events are causal for a property violation. These approaches require that the user has some intuition on which events could potentially be causal and, consequently, the completeness of the results depends on the complete identification of potential causal events by the user. The symbolic approach for fault tree generation presented in [24, 25] checks whether some events given as an input are causal for a property violation. The deductive cause-consequence analysis presented in [88] can be used to deduce the cause-consequence relationship between a predefined set of potentially causal events and a property violation. A combination of traditional manual safety analysis methods and causal reasoning is presented in [42].

The work in [12] is based on the causality checking approach proposed in this thesis and extends the causality checking algorithm in a way that it can be integrated into a bounded model checker based on Boolean satisfiability (SAT) solving. Since

the approach in [12] is based on bounded model checking [20] the results are only complete and sound with respect to the predefined bound k.

2.2 Probabilistic Causality Reasoning

As already discussed, the debugging of the probabilistic system models used in probabilistic model checking becomes even more difficult, since in almost all cases a set of error traces is needed to form a probabilistic counterexample [5, 47].

Work in [47] documents how probabilistic counterexamples for discrete-time Markov chains can be represented by regular expressions. While the regular expressions define an equivalence class for some traces in the counterexample, there is the possibility that not all possible traces are represented by the regular expression and, consequently, not all causal event combinations are captured by the regular expression.

In [5, 101] probabilistic counterexamples are represented by identifying a portion of an analyzed Markov chain in which the probability to reach a safety-critical state exceeds the probability bound specified by an upper-bounded reachability property. The method proposed in this thesis improves these approaches by identifying not only a portion of the Markov chain, but all causal event combinations and their corresponding order. Aljazzar et al. propose in [4] an interactive visualization of probabilistic counterexamples that while it facilitates the analysis of probabilistic counterexamples, still requires a manual inspection of a large number of error traces.

The approach of [22] computes minimal-cut sets, which are minimal combinations of events that are causal for a property violation, and their corresponding probabilities. Our approach extends and improves this approach by considering the event order as a causal factor.

In addition none of the approaches in [4, 5, 22, 47, 101] is able to reveal that the non-occurrence of an event is causal.

In [23], a formalization of the semantics of dynamic fault trees [37] and a probabilistic analysis framework for dynamic fault trees based on interactive Markov chains [48] is presented. The approach in [23] takes the dynamic fault tree as an input. As a consequence, while this approach allows for a probabilistic analysis of the events in the dynamic fault tree, no causality computation is performed and the user has to specify the causal event combinations in the form of a dynamic fault tree. Furthermore, there is no possibility to combine the analysis with a model containing the events of the dynamic fault tree.

In [34, 35] the approach from [17, 18] (cf. Section 2.1) is applied to discrete-time Markov chains and continuous-time Markov chains. The approach focuses on single counterexamples and the causality of variable value-changes for the violation of subformulas of the property, and suffers from the same shortcomings as the approach from [17, 18].

2.3 Conclusion

While there is ample research on using various notions of causality to aid in debugging and explaining single counterexamples, there does not exist an approach that identifies all causal combinations of events for a property violation and takes the causality of the event ordering for the property violation as well as the causality of the non-occurrence of events into account. In the debugging of a single error trace obtained from a sequential model this restriction might not restrain the analysis, since often the property violation can be traced back to one single causal event. In concurrent safety-critical systems consisting of both hardware and software it is usually the case that a property violation can not be caused by a single event, but instead requires multiple events to occur, sometimes even in a specific order. As a consequence, there is a need for a formal causality reasoning approach that allows for the analysis of concurrent safety-critical systems.

The objective of this thesis is to address this issue and to develop a novel approach for causality reasoning in concurrent system models that allows for the identification of all causal event combinations for a property violation. It shall be possible to consider complex event combinations consisting of disjunctions or conjunctions of event combinations or of single events as causes. Furthermore, the causality of the event orderings for a property violation shall be taken into account and it shall be detected if the non-occurrence of an event is causal for the property violation. The proposed method shall be applicable to both qualitative model checking and probabilistic model checking.

Chapter 3

Foundations

Contents

3.1 Introduction

In this chapter we briefly introduce into the foundations on which causality checking is build upon. We first present in Section 3.2 the running example of a railroad crossing that we use throughout the thesis to illustrate the proposed concepts. In Section 3.3 we briefly introduce the formal system model used for quantitative model checking and the logic used to formalize the requirements that shall be analyzed with the model checker. Furthermore, we introduce the Promela language which is the input language of the model checker that we use in this thesis. Probabilistic model checking and its underlying system model, as well as the logic used for the specification of probabilistic requirements and the notion of probabilistic counterexamples, are introduced in Section 3.4. In Section 3.4 we also introduce the input language of the probabilistic model checker that we use in this thesis. Fault trees which we will use in Chapter 10 to visualize the causality checking results are introduced in Section 3.5. In Chapter 4 we define a translation from event order logic formula to alternating automata. Alternating automata are introduced in Section 3.6. Finally, in Section 3.7 we introduce the QuantUM approach for model-based safety analysis. In Chapter 11 we discuss the integration of causality checking into the QuantUM approach.

3.2 Running Example: Railroad Crossing

We will demonstrate the definitions presented in this thesis on a running example of a railroad crossing system. In the running example a train can approach the

crossing, cross the crossing and finally leave the crossing. Whenever a train is approaching, the gate should close and will open when the train has left the crossing. It might also be the case that the gate fails. The car approaches the crossing and crosses the crossing if the gate is open and finally leaves the crossing. 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. In the following we will use the event identifiers defined in Table 3.1 to identify the events.

Identifier	Event
Ta	Train is approaching
Tc	Train is on the crossing
Tl	Train left the crossing
Gc	Gate is closed
Go	Gate is open
Gf	Gate failed
Ca	Car is approaching
Cc	Car is on the crossing
Cl	Car left the crossing

Table 3.1: Event identifiers of the railroad example.

3.3 Model Checking

Model checking [31] is an established technique for the verification of system models. For a formal model of the system and a formalized property the model checker automatically checks whether the model satisfies the property. In case the property is not satisfied, a trace from the initial system state into a state violating the property is produced by the model checker. This error trace is called a counterexample. Counterexamples can be used to retrace the steps of the system that lead to a particular property violating state.

The system models we use in this thesis for the model checking are transition systems which are introduced in Section 3.3.1. For the formalization of the requirements we use the linear temporal logic which is introduced in Section 3.3.2.

In this thesis we use the SpinJa [33] model checker, a Java re-implementation of the explicit state model checker Spin [50]. We use SpinJa instead of Spin, because SpinJa is written in an object-oriented programming language and its interfaces are cleanly engineered and thus it can be more easily to extend The input language of Spin which is also used by SpinJa is called Promela [50] which we briefly introduce in Section 3.3.3.

3.3.1 Transition System

The systems that we apply causality checking to are concurrent systems. For the formalization of the system model we follow the formalization of a model for concurrent computing systems proposed in [8]. The system model is given by a Transition

System which is defined as follows:

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

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

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

In the following we will use short-hand notation σ = "a_{α_1},a_{α_2}, ... , a_{α_n}" for an execution trace $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ ...\ \alpha_n\ s_n$. The trace σ = "Ta, Ca, Gf, Cc, Tc", for instance, is a trace of the railroad example from Sec. 3.2 where the train and the car are approaching the crossing (Ta, Ca), the gate fails to close (Gf), the car crosses the crossing (Cc) and finally the train crosses the crossing (Tc).

3.3.2 Linear Temporal Logic

Causality checking aims at identifying the causal events for the violation of functional safety requirements. Such a violation is also referred to as a hazard. We use linear time temporal logic (LTL) using the syntax and semantics as defined in [82] in order to specify hazards.

We use $T \vDash_l \varphi$ to express that the LTL formula φ holds for the transition system T and $\sigma \vDash_l \varphi$ respectively for execution traces.

Definition 3. *Syntax of LTL. Formulas in LTL over the set AP of atomic proposition are formed according to the following grammar given in BNF:*

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\ \varphi \mid \varphi_1\ \mathcal{U}\ \varphi_2$$

where $a \in AP$*. Additionally the following operators are defined as syntactic sugaring of those above:*

$$\Diamond\varphi = true\ \mathcal{U}\ \varphi \text{ and } \Box\varphi = \neg \Diamond \neg\varphi$$

Definition 4. *Semantics of LTL over Executions and States. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *be a transition system, let* φ *an LTL formula over AP and* σ *a finite execution of* T *and* $\sigma[j...]$ *the suffix of* σ *starting at* s_j*. Then the semantic is defined by induction on the structure of* φ

- $\sigma \vDash_l true$

- $\sigma \not\models_l \mathit{false}$
- $\sigma \models_l p$ *iff* $p \in L(s_0)$
- $\sigma \models_l \neg\varphi$ *iff* $\sigma \not\models_l \varphi$
- $\sigma \models_l \varphi_1 \wedge \varphi_2$ *iff* $\sigma \models_l \varphi_1$ *and* $\sigma \models_l \varphi_2$
- $\sigma \models_l \varphi_1 \vee \varphi_2$ *iff* $\sigma \models_l \varphi_1$ *or* $\sigma \models_l \varphi_2$
- $\sigma \models_l \bigcirc \varphi$ *iff* $\sigma[1...] \models_l \varphi$
- $\sigma \models_l \varphi_1 \mathcal{U} \varphi_2$ *iff* $\exists k \geq 0 \,.\, \sigma[k...] \models_l \varphi_2$ *and* $\forall 0 \leq j < k \,.\, \sigma[j...] \models_l \varphi_1$

and for the derived operators $\Diamond$ *and* $\Box$*.*

- $\sigma \models_l \Diamond\varphi$ *iff* $\exists j : j \geq 0 \,.\, \sigma[j..] \models_l \varphi$
- $\sigma \models_l \Box\varphi$ *iff* $\forall j : j \geq 0 \,.\, \sigma[j..] \models_l \varphi$

Hazards imply the reachability of unsafe states and they hence belong to the class of reachability properties. An unsafe state is reachable if a finite path satisfying the LTL property $\Diamond \mathit{unsafe}$ exists. For a system to be safe we require the non-reachability of unsafe states, which is expressed by the LTL formula $\Box\neg \mathit{unsafe}$. A counterexample for a non-reachability property is a finite execution trace fragment from the initial state to the unsafe state. Hence we only need to consider finite execution fragments [8]. Non-reachability properties fall within the class of safety properties in the commonly used classification scheme of safety and liveness properties [69].

The non-reachability of the hazard in our railroad example can be characterized by the LTL formula $\varphi = \Box\neg(\mathit{car\_crossing} \wedge \mathit{train\_crossing})$.

We can partition the set of all possible execution traces Σ of a transition system T into the set of "good" execution traces, denoted Σ_G, where the LTL formula characterizing the non-reachability of the hazard is not violated and the set of "bad" execution traces, denoted Σ_B, where the LTL formula characterizing the non-reachability of the hazard is violated. The elements of Σ_B are also referred to as counterexamples in model checking. The trace σ = "Ta, Ca, Gf, Cc, Tc" we already discussed above is a "bad" execution trace, since both the car and the train are on the crossing at the same time and thus the LTL property is violated. An example for a "good" trace is σ' ="Ta, Ca, Gf, Cc, Cl, Tc" where the car leaves the crossing (Cl) before the train is crossing (Tc) and, consequently, the train and the car are not on the crossing at the same time and the LTL formula is not violated.

Definition 5. *Good and Bad Execution Traces. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *be a transition system, let* φ *an LTL formula over* AP *and* Σ *that set of all possible finite executions of* T*. The set* Σ *is divided into into the set of "good" execution traces* Σ_G *and in the set of "bad" execution traces* Σ_B *as follows:* $\Sigma_G = \{\sigma \in \Sigma \mid \sigma \models_l \varphi\}$*,* $\Sigma_B = \{\sigma \in \Sigma \mid \sigma \not\models_l \varphi\}$ *and* $\Sigma_G \cup \Sigma_B = \Sigma$ *and* $\Sigma_G \cap \Sigma_B = \varnothing$*.*

3.3.3 The Promela Language

We briefly introduce the Promela language used as an input language for the Spin and SpinJa model checkers. For an in-depth introduction to Promela we refer to [50]. The Promela language is a C-like language that supports asynchronous communication as well as synchronous communication via rendezvous channels and synchronization via shared variables. A Promela model is composed of concurrent processes modeled by *proctypes*. If a process is marked as being an *active proctype* it is automatically started when the model is executed. In Figure 3.1 an example of a Promela proctype is given. The proctype comprises one boolean variable with the name *var1* which is initially false and one integer variable named *var2* which is initially 0. The do-loop is executed as long as the Promela model is executed. If the guard (var2 < 4) evaluates to true, *var2* will be incremented by 1. If the guard (var2 = 2) evaluates to true, *var1* will be set to true. In case two guards in a do-loop evaluate to true at the same time a non-deterministic choice will be made. The *atomic* command ensures that all commands enclosed by the atomic block are executed atomically which means that they are executed at the same time as if they were a single statement.

```
1  active proctype procA()
2  {
3      bool var1 = false;
4      int var2 = 0;
5
6      do
7      :: atomic{(var2 < 4) -> var2 = var2 + 1};
8      :: atomic{(var2 = 4) -> var1 = true};
9      od;
10 }
```

Listing 3.1: Example Promela code.

3.4 Probabilistic Model Checking

Probabilistic model checking [64] is an established automated analysis technique used amongst others in the analysis of safety-critical systems.

Probabilistic model checking requires two inputs:

- a description of the system to be analyzed, typically given in some model checker specific modeling language and
- a formal specification of quantitative properties of the system, relating for example to its performance or reliability that are to be analyzed.

From the first of these inputs, a probabilistic model checker constructs the corresponding probabilistic model. The probabilistic system models which we use in

this thesis are labeled continuous-time Markov chains which are introduced in Section 3.4.1. For the formalization of the requirements we use the continuous stochastic logic which is introduced in Section 3.4.2. The probabilistic model checker constructs the state space of the model in an exhaustive fashion, based on a systematic exploration of all possible states that can occur. In contrast to, for instance discrete-event simulation techniques [94] or statistical model checking[103], which generate approximate results by averaging results from a large number of random samples, probabilistic model checking applies numerical computation to yield exact results. The notion of probabilistic counterexamples that we use in this thesis is introduced in Section 3.4.3.

We use the probabilistic model checker PRISM [65], which is an open-source tool developed at the University of Oxford. The input language of PRISM, which is also called PRISM, is introduced in Section 3.4.4.

3.4.1 Labeled Continuous-time Markov Chain

The probabilistic system model used in probabilistic model checking is a probabilistic variant of a state-transition system, where each state represents a possible configuration of the system being modeled and each transition represents a possible evolution of the system from one configuration to another over time. The transitions are labeled with quantitative information specifying the probability and/or timing of the transition's occurrence. In the case of continuous-time Markov chain (CTMC) [59], which we use in this thesis, transitions are assigned positive, real values that are interpreted as the rates of negative exponential distributions.

Definition 6. *Labeled Continuous-time Markov Chain (CTMC). A labeled continuous-time Markov Chain C is a tuple $(\mathcal{S}, s_0, \mathcal{R}, \mathcal{L})$, where $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ is a transition rate matrix and $\mathcal{L} : \mathcal{S} \to 2^{AP}$ is a labeling function, which assigns to each state a subset of the set of atomic propositions AP.*

Since the notion of counterexample that we introduce in Section 3.4.3 is defined for discrete-time Markov chains (DTMC), we introduce the uniformization method [7] that is used to turn a CTMC into a DTMC. The uniformization method turns a given CTMC $C = (\mathcal{S}, s_0, \mathcal{R}, \mathcal{L})$ into a discrete-time Markov chain (DTMC) $D_C = (\mathcal{S}, s_0, \mathcal{P}, \mathcal{L})$, called the *uniformized DTMC*, where $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{P} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ is a probability matrix and $\mathcal{L} : \mathcal{S} \to 2^{AP}$ is a labeling function, which assigns to each state a subset of the set of atomic propositions AP. The uniformized DTMC is embedded into a Poisson process. The probability matrix $\mathcal{P}$ of the uniformized DTMC q is obtained by normalizing all rates in $\mathcal{C}$ with respect to the uniformization rate q, which is a real number such that $q \geq max\{\ \Lambda(s) \mid s \in \mathcal{S}\ \}$, where $\Lambda(s)$ refers to the total exit rate of s, i.e. $\Lambda(s) = \sum_{s' \in \mathcal{S}} \mathcal{R}(s, s')$. This means, the transition probability $\mathcal{P}(s,\ s') = \frac{\mathcal{R}(s,\ s')}{q}$ is assigned for each transition $(s,\ s')$. Each hop in the uniformized DTMC corresponds

to a time delay, which is exponentially distributed with a rate q. For each state s with $\Lambda(s) < q$, a self loop with the transition probability $\mathcal{P}(s,\ s) = \frac{q-\Lambda(s)}{q}$ is added.

3.4.2 Continuous Stochastic Logic

The quantitative properties of the system that are to be analyzed are specified using a variant of temporal logic. The temporal logic used for specifying properties over CTMCs is the Continuous Stochastic Logic (CSL) [1, 7]. This section provides a short introduction into CSL for a more comprehensive description we refer to [7]. CSL is a stochastic variant of the Computation Tree Logic (CTL) [30] with state and path formulas based on the work of Aziz et al. [6]. The state formulas are interpreted over states of a CTMC, whereas the path formulas are interpreted over paths in a CTMC. CSL extends CTL with two probabilistic operators that refer to the steady state and transient behavior of the model. The steady-state operator refers to the probability of residing in a particular set of states, specified by a state formula, in the long run. The transient operator allows us to refer to the probability of the occurrence of particular paths in the CTMC. In order to express the time span of a certain path, the path operators until (U) and next (X) are extended with a parameter that specifies a time interval.

We use the standard syntax and semantics of CSL as defined in [7].

Definition 7. *Syntax of CSL. The syntax of a CSL formula is defined as follows:*

$$\phi\ :=\ true \mid false \mid a \mid \neg\ \phi \mid \phi\ \vee\ \phi \mid \phi\ \wedge\ \phi \mid \mathcal{P}_{\bowtie p}(\varphi),$$

where $\bowtie$ is one of the operators $<$, $\leq$, $>$ or $\geq$, $p \in [0, 1]$ and φ is a path formula the syntax of which is defined as:

$$\varphi\ :=\ \phi\ U\ \phi \mid \phi\ U^{\leq t}\phi,$$

where $t \in \mathbb{R}_{\geq 0}$ is a non-negative real number.

If a path σ of a CTMC C fulfills the CSL formula φ we denote this by $\sigma \vDash_{\text{CSL}} \varphi$.

Since we want to compute the probability for a specific hazard to occur, the CSL property characterizes the reachability of the hazard. This is in contrast to the qualitative model checking where we use an LTL property characterizing the non-reachability of the hazard.

3.4.3 Probabilistic Counterexamples

Given an appropriate system model and a CSL property, probabilistic model checking tools such as PRISM can verify automatically whether the model satisfies the property. Probabilistic model checkers do not automatically provide counterexamples, but the computation of counterexamples has recently been addressed in, amongst others, [5, 47].

For the purpose of this thesis it suffices to consider only upper bounded probabilistic timed reachability properties. They require that the probability of reaching

a certain state, often corresponding to an undesired system state, does not exceed a certain upper probability bound p. In CSL such properties can be expressed by formulas of the form $\mathcal{P}_{\leq p}(\varphi)$, where φ is a path formula specifying undesired behavior of the system. A counterexample for an upper bounded property is a set Σ_C of paths leading from the initial state to a state satisfying φ such that the accumulated probability of Σ_C violates the probability constraint $\leq p$.

The probability of paths originating at the initial state of a Markov chain M is measurable by the probability measure Pr^M. In [8] is is shown that the underlying σ-algebra is formed by the cylinder sets which are induced by finite paths in $\mathcal{M}$ starting at s_0. The cylinder set spanned by the finite path fragments in $\mathcal{M}$ is defined as follows.

Definition 8. *Cylinder Set. Each finite path $s_0, \ldots, s_n$ induces a cylinder set $cyl(s_0, \ldots, s_n) = \{\sigma \in Paths^M(s_0) \mid \sigma^{(n)} = s_0, \ldots, s_n\}$.*

The probability of this cylinder set is defined in [8] as follows.

Definition 9. *Probability of a Cylinder Set.*

$$Pr^M(Cyl(s_0 \ldots s_n)) = P(s_0 \ldots s_n)$$

where

$$P(s_0 \ldots s_n) = \prod_{0 \leq i < n} P(s_i, s_{i+1})$$

and for path fragments of length zero let $P(s_0) = 1$.

The probability measure $Pr^M(X)$ for a set X of diagnostic paths is defined as

Definition 10. *$Pr^M(X)$ for a set X of diagnostic paths.*

$$Pr^M(X) \quad = \sum_{\sigma \in X} P(\sigma)$$

A probabilistic counterexample is defined in [5] as follows.

Definition 11. *Probabilistic Counterexample. Let M be a Markov chain which violates the upper-bounded formula $\mathcal{P}_{\lhd p}(\varphi)$, where $\lhd$ is either $<$ or $\leq$. A **probabilistic counterexample** of $\mathcal{P}_{\lhd p}(\varphi)$ is a subset of diagnostic paths, i.e. $X \subseteq \{\, \sigma \in Paths^M(\hat{s}) \mid \sigma \vDash_{CSL} \varphi \,\}$, which is measurable and for which $Pr^M(X) \lhd p$ does not hold.*

If the CSL formula $\mathcal{P}_{=?}(\varphi)$ is used, the probability of the path formula φ to hold is computed and the counterexample contains all paths fulfilling φ. The probability of the counterexample is computed using a probabilistic model checker, in our case PRISM. Notice that in the setting of this thesis the counterexample is computed completely, i.e., all paths leading into the undesired system state are enumerated in the counterexample.

3.4.4 The PRISM Language

We present an overview of the input language of the probabilistic model checker PRISM [65], for a precise definition of the semantics refer to [49]. In Section 8.2 we present a mapping from probabilistic PRISM models to qualitative Promela models. A PRISM model is composed of a number of *modules* which can interact with each other. A *module* contains a number of local variables. The values of these variables at any given time constitute the state of the *module*. The global state of the entire model is determined by the local state of all *modules*. The behavior of each module is described by a set of commands. A command takes the form: "[*action_label*] *guard* → $rate_1$: $update_1$&...& $update_n$;". The *guard* is a predicate over all variables in the model. The *update* commands describe a transition which the module can make if the *guard* is true. A transition is specified by giving the new values of the variables in the *module*, possibly as a function of other variables. A *rate* is assigned to each transition. The *action_label* is used for synchronizing transitions of different modules. If two transitions are synchronized they can only be executed if the guards of both transitions evaluate to true. The rate of the resulting synchronized transition is the product of the two individual transitions. An example of a PRISM model is given in Listing 3.2. The module named *moduleA* contains two variables: *var1*, which is of type Boolean and is initially *false*, and *var2*, which is a numeric variable and has initially the value 0. If the guard $(\text{var2} < 4)$ evaluates to true, the update $(\text{var2}' = \text{var2} + 1)$ is executed with a rate of 0.8. If the guard $(\text{var2} = 2)$ evaluates to true, the update $(\text{var1}' = \textit{true})$ is executed with a rate of 1.0.

```
1  module moduleA
2     var1: bool init false;
3     var2: [0..11] init 0;
4     [Count] (var2 < 4) -> 0.8: ( var2'= var2 + 1);
5     [End] (var2 = 4) -> 1.0: ( var1'= true);
6  endmodule
7  module moduleB
8     var3: [0..2] init 0;
9     [Count] (var3 < 2) -> 1.0: ( var3'= var3 + 1);
10    [Count] (var3 = 2) -> 1.0: ( var3'= 0);
11 endmodule
```

Listing 3.2: Example code in the PRISM language.

The transitions with the action label *Count* of the modules moduleA and moduleB are synchronized. If the guard of the transition of *moduleA* labeled with *Count* evaluates to true and one of the guards of the transitions of *moduleB* labeled with *Count* evaluates to true, then the transition of moduleA will be executed simultaneously with the transition of moduleB for which the guard evaluates to true. If the guard of the transition of *moduleA* labeled with *Count* evaluates to true and both of the guards of the transitions of *moduleB* evaluate to true, one of the transitions

of *moduleB* will be selected by a stochastic race and executed simultaneously with the transition of *moduleA*. If only the guard of the transition in *moduleA* labeled with *Count* evaluates to true, or only the guards of one or both of the transition in *moduleB* labeled with *Count* evaluate to true no transition will be executed.

3.5 Fault Trees

Fault trees (FTs) [97] are being used extensively in industrial practice, in particular in fault prediction and analysis, to illustrate graphically under which conditions systems can fail, or have failed. We use fault trees in Chapter 10 in order to provide a graphical representation of the causality checking results.

In the context of this thesis, we need the following elements of fault trees:

1. Basic event: represents an atomic event. In the context of this work an atomic event is the transition from one state in the transition system to another state.
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 or specified by an order constraint connected to the *PAND*-gate.
5. Intermediate Event: failure events that are caused by their child nodes. The probability of the intermediate event to occur is denoted by the number in the lower right corner. 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. 3.1. The *AND*, *OR* and *PAND* gates are used to express that their top events are caused by their input events.

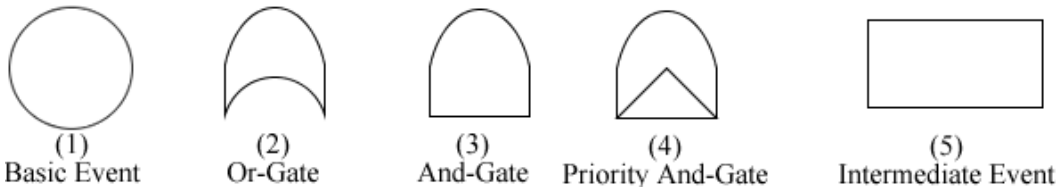


Figure 3.1: Fault Tree Elements.

Figure 3.2 shows an example fault tree of the railroad crossing example introduced in Section 3.2. The top level event represents the cases where both the car and the train are on the crossing. There are two possibilities for the TLE to happen, which are represented by the two branches of the fault tree that are connected with the TLE by an OR-gate. The events of each of the two branches are connected by an AND-gate, which means that all of the events have to occur in order for the

TLE to occur. Note that no information on the order of the events is given in the example.

1. If both a train (Ta) and a car (Ca) are approaching and the gate fails this results in a hazardous situation if the car is on the crossing (Cc) and does not leave the crossing (Cl) and the train (Tc) enters the crossing.

2. If both a train (Ta) and a car (Ca) are approaching and the gate closes (Gc) when the car (Cc) is already on the railroad crossing and is not able to leave (Cl) and the train is crossing (Tc), this also corresponds to a hazardous situation.

For an in-depth discussion of fault trees we refer the reader to [97].

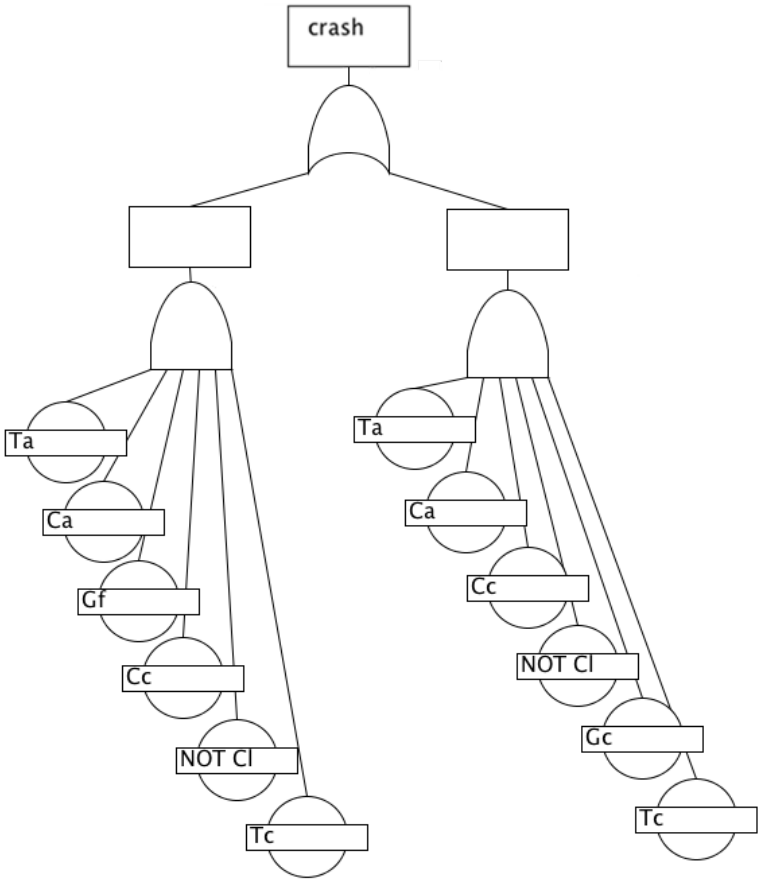


Figure 3.2: Example fault tree of the railroad crossing example.

3.6 Alternating Automata

Alternating automata [28, 98] are a generalization of nondeterministic automata in which choices along a path can be marked existential, meaning that some path has

to reach an accepting state, or universal, which means that all paths have to reach an accepting state.

In Chapter 4 we show how formulas in the event order logic proposed in this thesis can be translated into alternating automata, which we use in Chapter 8 in order to speed up the probability computation for causal event combinations.

We use the definition of alternating automata from [83], which differs from the definitions in [28, 98] in that the automata are not defined with input symbols labeling the edges but with input symbols labeling the nodes instead. We use the definition from [83] because a graphical representation for alternating automata defined according to [83] exists.

Definition 12. *Alternating Automaton. An alternating automaton A is defined recursively as follows:*

$$\begin{array}{ll} A ::= \epsilon_A & \textit{(empty automaton)} \\ \mid \langle v, \delta, f \rangle & \textit{(single node)} \\ \mid A_1 \wedge A_2 & \textit{(conjunction of two automata)} \\ \mid A_1 \vee A_2 & \textit{(disjunction of two automata)} \end{array}$$

where v is a state formula, which can be either a propositional or a first-order formula, δ is an alternating automaton expressing the next-state relation, and f indicates whether the node is accepting (denoted by +) or rejecting (−). We require the automaton to be finite.

The set of nodes of an automaton A, denoted by $\mathcal{N}(A)$ is formally defined as

$$\begin{array}{ll} \mathcal{N}(\epsilon_A) & = \varnothing \\ \mathcal{N}(\langle v, \delta, f \rangle) & = \langle v, \delta, f \rangle \cup \mathcal{N}(\delta) \\ \mathcal{N}(A_1 \wedge A_2) & = \mathcal{N}(A_1) \cup \mathcal{N}(A_2) \\ \mathcal{N}(A_1 \vee A_2) & = \mathcal{N}(A_1) \cup \mathcal{N}(A_2) \end{array}$$

We denote with $\mathcal{N}_{rej}(A)$ the set of nodes of A that are rejecting, that is

$$\mathcal{N}_{rej}(A) = \{n \in \mathcal{N}(A) | f(n) = -\}$$

.

A path through an ω-automaton is an infinite sequence of nodes. A "path" through an alternating ω-automaton is, in general, a tree.

Definition 13. *Tree. A tree is defined recursively as follows:*

$$\begin{array}{ll} T ::= \epsilon_T & \textit{(empty tree)} \\ \mid T \cdot T & \textit{(composition)} \\ \mid \langle \langle v, \delta, f \rangle, T \rangle & \textit{(single node with child tree)} \end{array}$$

A tree may have both finite and infinite branches.

The alternating automaton $A_1 = \langle true, A_2 \wedge \langle a, A_3, + \rangle, + \rangle$ where $A_2 = \langle b, A_2, + \rangle$ and $A_3 = \langle true, A_3, + \rangle$ can be graphically represented by the automaton in Figure 3.3. The arc between two edges denotes an "and" choice where both paths have to be accepting. The absence of the arc would denote an "or" choice, where only one of the paths has to be accepting. The automaton A_1 accepts only sequences of the form $\langle ?, ? \rangle, \langle b, a \rangle, \langle b, ? \rangle, \langle b, ? \rangle, ...$, where ? denotes a "don't care":

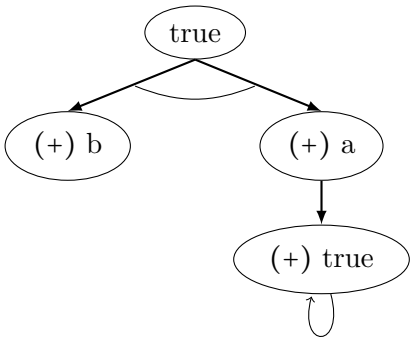


Figure 3.3: Graphical representation of the alternating automaton $A_1 = \langle true, A_2 \wedge \langle a, A_3, + \rangle, + \rangle$ where $A_2 = \langle b, A_2, + \rangle$ and $A_3 = \langle true, A_3, + \rangle$.

Definition 14. *Run of an Alternating Automaton. Given an infinite sequence of states $\sigma = s_0, ..., s_{n-1}$ and an automaton A, a tree T is called a run of σ in A if one of the following holds:*

$A = \epsilon_A$	*and*	$T = \epsilon_T$
$A = \langle v, \delta, f \rangle$	*and*	$n > 1$, $T = \langle \langle v, \delta, f \rangle, T' \rangle$, $s_0 \vDash v$ *and* T' *is a run of* $s_1, ..., s_{n-1}$ *in* δ, *or* $n = 1$, $T = \langle \langle v, \delta, f \rangle, \epsilon_T \rangle$ *and* $s_0 \vDash v$
$A = A_1 \wedge A_2$	*and*	$T = T_1 \cdot T_2$, *which means that* T *is the composition of* T_1 *and* T_2 *where* T_1 *is a run of* A_1 *and* T_2 *is a run of* A_2
$A = A_1 \vee A_2$	*and*	T *is a run of* A_1 *or* T *is a run of* A_2

Definition 15. *Finite Accepting Run. A finite run T is accepting if every path through the tree ends in an accepting node.*

Definition 16. *Infinite Accepting Run. A infinite run T is accepting if every infinite branch contains infinitely many accepting nodes.*

A run of the sequence $\langle b, a \rangle, \langle b, a \rangle, \langle b, \neg a \rangle, \langle b, a \rangle, ...$ in the automaton A_1 in Figure 3.3 is shown in Figure 3.4. Both branches of the tree contain infinitely many accepting nodes and thus the run is accepting.

Alternating ω-automata can be used to represent temporal formulas. It was shown in [99, 84] that each LTL formula can be translated into an alternating ω-automaton.

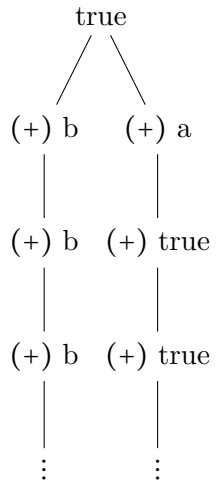


Figure 3.4: Run of the sequence $\langle b, a\rangle, \langle b, a\rangle, \langle b, \neg a\rangle, \langle b, a\rangle, ...$ in the automaton A_1.

Definition 17. *Alternating Automaton for an LTL formula. Given an LTL formula φ, an alternating automaton $A(\varphi)$ is constructed, as follows.*

For a state formula p:

$$A(p) = \langle p, \epsilon_A, + \rangle$$

For LTL formulas φ, φ_1, and φ_2:

$$\begin{aligned}
A(\varphi_1 \wedge \varphi_2) &= A(\varphi_1) \wedge A(\varphi_2) \\
A(\varphi_1 \vee \varphi_2) &= A(\varphi_1) \vee A(\varphi_2) \\
A(\bigcirc\varphi) &= \langle true, A(\varphi), + \rangle \\
A(\Box\varphi) &= \langle true, A(\Box\varphi), + \rangle \wedge A(\varphi) \\
A(\Diamond\varphi) &= \langle true, A(\Diamond\varphi), - \rangle \vee A(\varphi) \\
A(\varphi_1 \mathcal{U} \varphi_2) &= A(\varphi_2) \vee (\langle true, A(\varphi_1 \mathcal{U} \varphi_2), - \rangle \wedge A(\varphi_1))
\end{aligned}$$

Example 1. *Figure 3.5 shows the alternating automaton for the LTL formula $\Box a$.*

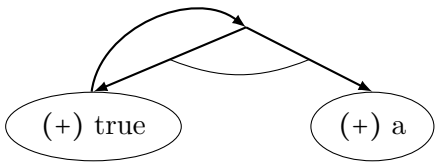


Figure 3.5: Graphical representation of the alternating automaton representing the LTL formula $\Box a$.

3.7 The QuantUM Approach

The QuantUM approach [71, 72] was introduced in order to close the gap between modern model-based development processes and state-of-the-art model checking techniques. We will briefly introduce the QuantUM approach here, a detailed description of QuantUM is given in [71, 72]. In Chapter 11 we discuss how the causality checking approach can be integrated into QuantUM. QuantUM supports the specification and analysis of system dependability requirements at the high level of architectural description that UML and SysML models offer. Using QuantUM, the information that is required as input for a safety analysis, like for instance failure modes and rates, can be specified directly in the UML/SysML models using stereotypes [86, 52]. Stereotypes contain meta information which is added to the model. The translation of the UML/SysML models to the PRISM language is fully automated. The PRISM model checker is encapsulated by QuantUM and its use is made fully transparent to the user, hence lowering the acceptance bar that formal methods often face in industrial engineering practice. Users do not have to understand the underlying formal verification technology in detail. A feature of QuantUM is that it can cooperate with many industrial practice UML CASE tools. The analysis in QuantUM is fully automated, since a manual analysis would be entirely infeasible, due to the inherent system complexity.

The QuantUM approach can be summarized by identifying the following steps:

1. The QuantUM UML/SysML profile extension is used to annotate the UML / SysML model with all information that is needed to perform a dependability analysis. The editing of the model can be performed in the standard industrial CASE tool which is used by the user.

2. The annotated UML model is exported in the XML Metadata Interchange [87] format, which is the standard format for exchanging UML models. The XMI format is supported by most CASE tools used in the industry.

3. Subsequently, the QuantUM tool parses the generated XMI file and generates an analysis model in the input language of the Spin model checker or the probabilistic model checker PRISM. QuantUM also generates a formal specification of the properties to be verified.

The QuantUM extension of the UML allows for a direct annotation of UML models within the case tool that is used to design the system. This allows for an convenient integration of the QuantUM approach into the development process of safety-critical systems.

Chapter 4

Event Order Logic

The content of this chapter is based on the publications [16, 61, 62, 78].

Contents

4.1 Introduction

In order to be able to reason about the causality of events we have to formally capture the occurrence of events. In order to do so we propose the *Event Order Logic (EOL)* which can be fully translated into LTL and allows to formally capture the order and occurrence of events on an execution trace of a transition system.

We propose the event order logic instead of using LTL because we need a logic to represent execution traces in as simple a way as possible. Furthermore, the EOL is used to represent the result of the causality checking process and, hence, needs to be easily understandable and the formulas should be as compact as possible.

As an alternative to the event order logic we also investigated the usage of the interval logics [93] and [36]. Given the fact that interval logics are overly expressive for the relatively simple ordering constraints, we decided to define our own tailored, relatively simple event order logic.

We define the syntax and semantics of the EOL in Section 4.2. In Section 4.3 we show that each EOL formula can be translated into an equivalent LTL formula. The relationship of EOL to ω-automata is discussed in Section 4.4.

4.2 Syntax and Semantics

We assume that for a given execution trace σ of a transition system T, the set Act of actions contains the events that we wish to reason about. For an LTL formula φ specifying a safety requirement and an execution trace σ, the hazard specified by the safety requirement occurs on σ if and only if $\sigma \not\models_l \varphi$ holds. Note that since each transition is only labeled with one action, only one event can occur per transition.

We assume that there exists a set $\mathcal{A}$ of event variables that contains a boolean variable a for each action $\alpha \in Act$ for some given transition system. The variable a_{Ta}, for instance, represents the event "train approaching the crossing".

We distinguish between event types that represent the type of an event and event occurrences which is the actual instance of an event type. The causality checking approach defined in this thesis will identify the event occurrences that are causal for a property violation. If multiple instances of one event type occur on one execution trace, the variables representing them are numbered according to their occurrence. For example, the two train approaching events on "Ta,Gc,Tc,Tl,Go,Ta" are numbered as a_{Ta_1} and a_{Ta_2}. The numbering of multiple instances of one event type is important in order to detect cases where an event of a specific event type has to occur more than once in order to cause the hazard. Consider the example of a system that will fail when two sensors have failed. The event of the type *sensor failure*, consequently, has to occur at least two times in order for the system to fail. If only event types would be considered for the causality analysis, it would not be possible to detect that the sensor failure has to occur at least two times.

In other words, the *i-th* occurrence of some action of type α will be represented by the boolean variable a_{α_i}. In the following we also abbreviate the event variable a_{Ta} by Ta.

Definition 18. *Events, Event Types and Event Variables. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system and* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a finite execution trace of T. We define the following: each* $\alpha \in Act$ *defines an* event type α. α_i *of* σ *denotes the* i-th *occurrence of an event of the* event type α. *The variable representing the occurrence of the event* α_i *is denoted by* a_{α_i}, *and the set* $\mathcal{A} = \{a_{\alpha_1}, ..., a_{\alpha_n}\}$ *contains a boolean variable for each occurrence of an event.*

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

Definition 19. *Valuation of the Set of Event Variables. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system,* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a finite execution trace of T and* $\mathcal{A}$ *the set of event variables then we define the function* $val_{\mathcal{A}}(\sigma)$ *as follows:*

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

Further we define $val_{\mathcal{A}}(\sigma) = val_{\mathcal{A}}(\sigma')$ *if for all* $a_{\alpha_i} \in \mathcal{A}$ *the values assigned by* $val_{\mathcal{A}}(\sigma)$ *and* $val_{\mathcal{A}}(\sigma')$ *are equal and* $val_{\mathcal{A}}(\sigma) \neq val_{\mathcal{A}}(\sigma')$ *else.*

Event variables allow us to reason about the occurrence of single events, but since we want to reason about the combination of events, we need a formalism that allows us to express the occurrence of event combinations. The event order logic

allows one to connect event variables from $\mathcal{A}$ with the boolean connectives $\wedge$, $\vee$ and $\neg$. To express the ordering of events we introduce the ordered conjunction operator $\barwedge$. The formula $a \barwedge b$ with events a and b is satisfied if and only if events a and b occur in a trace and a occurs before b. In addition to the $\barwedge$ operator we introduce the interval operators $\barwedge_{[}$, $\barwedge_{]}$, and $\barwedge_{<} \phi \barwedge_{>}$, which define an interval in which an event has to hold in all states. These interval operators are necessary to express the causal non-occurrence of events.

Definition 20. *Syntax of Event Order Logic (EOL). Simple EOL formulas over a set $\mathcal{A}$ of event variables are formed according to the following grammar:*

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \phi_1 \vee \phi_2$$

where $a \in \mathcal{A}$ and ϕ, ϕ_1 and ϕ_2 are simple EOL formulas. Complex EOL formulas are formed according to the following grammar:

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

where ϕ is a simple EOL formula and ψ_1 and ψ_2 are complex EOL formulas. Note that the $\neg$ operator binds more tightly than the $\barwedge$, $\barwedge_{[}$, $\barwedge_{]}$, and $\barwedge_{<} \phi \barwedge_{>}$, operators and those bind more tightly than the $\vee$ and $\wedge$ operator.

Note that only simple event order logic formulas can be negated. The negation of complex event order logic formulas is not allowed. Furthermore, De Morgan's laws do not apply to event order logic formulas as the semantics definition will show.

The formal semantics of the event order logic is defined over execution traces. Notice that the $\barwedge$, $\barwedge_{[}$, $\barwedge_{]}$, and $\barwedge_{<} \phi \barwedge_{>}$ operators are linear temporal logic operators and that the execution trace σ is akin to a linearly ordered Kripke structure.

Definition 21. *Semantics of Event Order Logic (EOL). Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, let ϕ, ϕ_1, ϕ_2 simple EOL formulas, let ψ, ψ_1, ψ_2 complex EOL formulas, and let $\mathcal{A}$ a set of event variables, with $a_{\alpha_i} \in \mathcal{A}$, over which ϕ, ϕ_1, ϕ_2 are built. Let $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ a finite execution trace of T and $\sigma[i..r] = s_i, \alpha_{i+1}, s_{i+1}, \alpha_{i+2}, \ldots \alpha_r, s_r$ a partial trace. We define that an execution trace σ satisfies an event order logic formula ψ, written as $\sigma \vDash_e \psi$, as follows:*

$$s_j \vDash_e a_{\alpha_i} \text{ iff } s_{j-1} \xrightarrow{\alpha_i} s_j \tag{4.1}$$

$$s_j \vDash_e \neg\phi \text{ iff not } s_j \vDash_e \phi \tag{4.2}$$

$$\sigma[i..r] \vDash_e \phi \text{ iff } \exists j : i \leq j \leq r \,.\, s_j \vDash_e \phi \tag{4.3}$$

$$\sigma[i..r] \vDash_e \neg\phi \text{ iff } \forall j : i \leq j \leq r \,.\, s_j \vDash_e \neg\phi \tag{4.4}$$

$$\sigma \vDash_e \psi \text{ iff } \sigma[0..n] \vDash_e \psi, \text{ where } n \text{ is the length of } \sigma. \tag{4.5}$$

$$\sigma[i..r] \vDash_e \phi_1 \wedge \phi_2 \text{ iff } \sigma[i..r] \vDash_e \phi_1 \text{ and } \sigma[i..r] \vDash_e \phi_2 \tag{4.6}$$

$$\sigma[i..r] \vDash_e \phi_1 \vee \phi_2 \text{ iff } \sigma[i..r] \vDash_e \phi_1 \text{ or } \sigma[i..r] \vDash_e \phi_2 \tag{4.7}$$

$$\sigma[i..r] \vDash_e \neg(\phi_1 \wedge \phi_2) \text{ iff } \sigma[i..r] \vDash_e \neg\phi_1 \text{ and } \sigma[i..r] \vDash_e \neg\phi_2 \tag{4.8}$$

$$\sigma[i..r] \vDash_e \neg(\phi_1 \vee \phi_2) \text{ iff } \sigma[i..r] \vDash_e \neg\phi_1 \text{ and } \sigma[i..r] \vDash_e \neg\phi_2 \tag{4.9}$$

$$\sigma[i..r] \vDash_e \psi_1 \wedge \psi_2 \text{ iff } \sigma[i..r] \vDash_e \psi_1 \text{ and } \sigma[i..r] \vDash_e \psi_2 \tag{4.10}$$

$$\sigma[i..r] \vDash_e \psi_1 \vee \psi_2 \text{ iff } \sigma[i..r] \vDash_e \psi_1 \text{ or } \sigma[i..r] \vDash_e \psi_2 \tag{4.11}$$

$$\sigma[i..r] \vDash_e \psi_1 \barwedge \psi_2 \text{ iff } \exists j,k : i \leq j < k \leq r \,.\, \sigma[i..j] \vDash_e \psi_1 \text{ and } \sigma[k..r] \vDash_e \psi_2 \tag{4.12}$$

$$\sigma[i..r] \vDash_e \psi \barwedge_{[} \phi \text{ iff } (\exists j : i \leq j \leq r \,.\, \sigma[j..j] \vDash_e \psi \text{ and } (\forall k : j \leq k \leq r \,.\, \sigma[k..k] \vDash_e \phi)) \tag{4.13}$$

$$\sigma[i..r] \vDash_e \phi \barwedge_{]} \psi \text{ iff } (\exists j : i \leq j \leq r \,.\, \sigma[j..j] \vDash_e \psi \text{ and } (\forall k : 0 \leq k \leq j \,.\, \sigma[k..k] \vDash_e \phi)) \tag{4.14}$$

$$\sigma[i..r] \vDash_e \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \text{ iff } (\exists j,k : i \leq j < k \leq r \,.\, \sigma[j..j] \vDash_e \psi_1 \text{ and } \sigma[k..r] \vDash_e \psi_2 \text{ and } (\forall l : j \leq l \leq k \,.\, \sigma[l..l] \vDash_e \phi)) \tag{4.15}$$

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

Note that the event order logic semantics that we have defined is an existential semantics. Consequently, if there exists one trace of a transition system that satisfies an event order logic formula, the event order logic formula holds for the transition system. The existential semantics is sufficient for causality checking since we want to formally capture the existence of event combinations and event orders.

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

Definition 22. *Event Order Logic (EOL) Formula over Executions. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and $\sigma = s_0\ \alpha_1\ s_1\ \alpha_2\ ...\ \alpha_n\ s_n$ an execution trace of T. The event order logic formula over the execution σ denoted by ψ_σ is defined as follows: We partition the set $\mathcal{A}$ into the set Z and the set W such that $Z \subseteq \mathcal{A}$, $W \subseteq \mathcal{A}$, $Z \cap W = \varnothing$, $Z \cup W = \mathcal{A}$ and $\forall a_{\alpha_i} \in Z : \sigma \vDash_e a_{\alpha_1}$ and $\forall a_{\alpha_j} \in W : \sigma \nvDash_e a_{\alpha_1}$. The EOL formula ψ_σ is formed over the event variables in Z by connecting the event variables with the $\barwedge$-operator $\psi_\sigma = a_{\alpha_i} \barwedge a_{\alpha_j} \barwedge ... \barwedge a_{\alpha_n}$ such that $i < j < n$ and $\sigma \vDash_e \psi_\sigma$.*

Furthermore, we define the subset operator for EOL formulas as follows.

Definition 23. *EOL Formula Subset Relationship. Let ψ_1 and ψ_2 EOL formulas, where ψ_1 is built over the set of event variables Z_1 and ψ_2 is built over the set of event variables Z_2.*

$\subseteq$**:** *$\psi_1 \subseteq \psi_2$ iff $Z_1 \subseteq Z_2$.*

$\subset$**:** *$\psi_1 \subset \psi_2$ iff $Z_1 \subseteq Z_2$ and not $Z_1 = Z_2$.*

We will now show that there are EOL formulas that are different with respect to syntax but equivalent with respect to the semantics and, consequently, evaluate to the same truth-value under all interpretations. These equivalences can be used to rewrite EOL formulas, which we will later need in order to translate EOL formulas into LTL formulas.

Definition 24. *Equivalences of EOL formulas. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, let* ϕ_1, ϕ_2 *simple EOL formulas, let* ψ_1, ψ_2, ψ_3 *complex EOL formulas, and let* $\mathcal{A}$ *a set of event variables, with* $a_{\alpha_i} \in \mathcal{A}$, *over which* ϕ, ϕ_1, ϕ_2 *are built. Let* $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ *a finite execution trace of* T *and* $\sigma[i..r] = s_i, \alpha_{i+1}, s_{i+1}, \alpha_{i+2}, \ldots \alpha_r, s_r$ *a partial trace. Two EOL formulas* ψ_1 *and* ψ_2 *are equivalent, denoted by* $\psi_1 \equiv \psi_2$ *iff* $\sigma[i..r] \vDash_e \psi_1 \Leftrightarrow \sigma[i..r] \vDash_e \psi_2$.

An example of two equivalent formulas from the railroad crossing example from Section 3.2 are $\psi_1 = \mathrm{Ta} \barwedge \mathrm{Tc} \wedge \mathrm{Ca} \barwedge \mathrm{Cc}$ and $\psi_2 = \mathrm{Ca} \barwedge \mathrm{Cc} \wedge \mathrm{Ta} \barwedge \mathrm{Tc}$ both of which state that the event train approaching (Ta) happens before train crossing (Tc) and car approaching (Ca) happens before car crossing (Cc) without imposing a restriction on the order of, for instance, train approaching and car approaching. Another example is $\psi_3 = (\neg\mathrm{Gc} \wedge \neg\mathrm{Gf}) \barwedge\rfloor \mathrm{Ca}$ and $\psi_4 = \neg\mathrm{Gc} \barwedge\rfloor \mathrm{Ca} \wedge \neg\mathrm{Gf} \barwedge\rfloor \mathrm{Ca}$ which both state that before the car approaching event neither the gate closing event (Gc) nor the gate failed (Gf) occurs.

Figure 4.1 shows some equivalences rules for EOL which are proven by Theorem 1 to Theorem 27.

Note that $\neg(\phi_1 \wedge \phi_2) \not\equiv (\neg\phi_1 \vee \neg\phi_2)$ since, for instance, the trace σ = "Ta, Ca" satisfies $(\neg\mathrm{Gf} \vee \neg\mathrm{Ta})$ but does not satisfy $\neg(\mathrm{Gf} \wedge \mathrm{Ta})$.

Theorem 1. $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$

Proof. $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \Leftrightarrow \sigma \vDash_e \neg\phi_1 \wedge \neg\phi_2$.

$$\begin{aligned} \sigma[i..r] \vDash_e \neg(\phi_1 \wedge \phi_2) \Leftrightarrow \quad & \sigma[i..r] \vDash_e \neg\phi_1 \text{ and } \sigma[i..r] \vDash_e \neg\phi_2 && \text{(Def. 21 (4.8))} \\ \Leftrightarrow \quad & \sigma[i..r] \vDash_e \neg\phi_1 \wedge \neg\phi_2 && \text{(Def. 21 (4.6))} \end{aligned}$$

□

Theorem 2. $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$

Proof. $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \neg(\phi_1 \vee \phi_2) \Leftrightarrow \sigma \vDash_e \neg\phi_1 \wedge \neg\phi_2$.

$$\begin{aligned} \sigma[i..r] \vDash_e \neg(\phi_1 \vee \phi_2) \Leftrightarrow \quad & \sigma[i..r] \vDash_e \neg\phi_1 \text{ and } \sigma[i..r] \vDash_e \neg\phi_2 && \text{(Def. 21 (4.9))} \\ \Leftrightarrow \quad & \sigma[i..r] \vDash_e \neg\phi_1 \wedge \neg\phi_2 && \text{(Def. 21 (4.6))} \end{aligned}$$

□

$$
\begin{array}{ll}
\neg(\phi_1 \wedge \phi_2) & \equiv \neg\phi_1 \wedge \neg\phi_2 \\
\neg(\phi_1 \vee \phi_2) & \equiv \neg\phi_1 \wedge \neg\phi_2 \\
\neg(\phi_1 \wedge \phi_2) & \equiv \neg(\phi_1 \vee \phi_2) \\
\psi_1 \wedge \psi_2 & \equiv \psi_1 \barwedge \psi_2 \vee \psi_2 \barwedge \psi_1 \\
(\psi_1 \wedge \psi_2) \barwedge \psi_3 & \equiv \psi_1 \barwedge \psi_3 \wedge \psi_2 \barwedge \psi_3 \\
\psi_1 \barwedge (\psi_2 \wedge \psi_3) & \equiv \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \\
(\psi_1 \vee \psi_2) \barwedge \psi_3 & \equiv \psi_1 \barwedge \psi_3 \vee \psi_2 \barwedge \psi_3 \\
\psi_1 \barwedge (\psi_2 \vee \psi_3) & \equiv \psi_1 \barwedge \psi_2 \vee \psi_1 \barwedge \psi_3 \\
\psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 & \equiv \psi_1 \barwedge \psi_3 \wedge \psi_1 \barwedge \psi_2 \\
\psi_1 \barwedge \psi_2 \barwedge \psi_3 & \equiv \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \wedge \psi_2 \barwedge \psi_3 \\
\psi_1 \barwedge_{[} (\phi_1 \wedge \phi_2) & \equiv \psi_1 \barwedge_{[} \phi_1 \wedge \psi_1 \barwedge_{[} \phi_2 \\
\psi_1 \barwedge_{[} (\phi_1 \vee \phi_2) & \equiv \psi_1 \barwedge_{[} \phi_1 \vee \psi_1 \barwedge_{[} \phi_2 \\
\psi_1 \barwedge \psi_2 \barwedge_{[} \phi & \equiv \psi_1 \barwedge \psi_2 \wedge \psi_2 \barwedge_{[} \phi \\
(\psi_1 \vee \psi_2) \barwedge_{[} \phi & \equiv \psi_1 \barwedge_{[} \phi \vee \psi_2 \barwedge_{[} \phi \\
(\psi_1 \wedge \psi_2) \barwedge_{[} \phi & \equiv \psi_1 \barwedge \psi_2 \barwedge_{[} \phi \vee \psi_2 \barwedge \psi_1 \barwedge_{[} \phi \\
(\phi_1 \wedge \phi_2) \barwedge_{]} \psi_1 & \equiv \phi_1 \barwedge_{]} \psi_1 \wedge \phi_2 \barwedge_{]} \psi_1 \\
(\phi_1 \vee \phi_2) \barwedge_{]} \psi_1 & \equiv \phi_1 \barwedge_{]} \psi_1 \vee \phi_2 \barwedge_{]} \psi_1 \\
\phi \barwedge_{]} \psi_1 \barwedge \psi_2 & \equiv \phi \barwedge_{]} \psi_1 \wedge \psi_1 \barwedge \psi_2 \\
\phi \barwedge_{]} (\psi_1 \vee \psi_2) & \equiv \phi \barwedge_{]} \psi_1 \vee \phi \barwedge_{]} \psi_2 \\
\phi \barwedge_{]} (\psi_1 \wedge \psi_2) & \equiv \phi \barwedge_{]} \psi_1 \barwedge \psi_2 \vee \phi \barwedge_{]} \psi_2 \barwedge \psi_1 \\
\psi_1 \barwedge_{<} (\phi_1 \vee \phi_2) \barwedge_{>} \psi_2 & \equiv \psi_1 \barwedge_{<} \phi_1 \barwedge_{>} \psi_2 \vee \psi_1 \barwedge_{<} \phi_2 \barwedge_{>} \psi_2 \\
\psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 & \equiv \psi_1 \barwedge \psi_2 \wedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \\
\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \barwedge \psi_3 & \equiv \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \wedge \psi_2 \barwedge \psi_3 \\
(\psi_1 \vee \psi_2) \barwedge_{<} \phi \barwedge_{>} \psi_3 & \equiv \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3 \vee \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \\
(\psi_1 \wedge \psi_2) \barwedge_{<} \phi \barwedge_{>} \psi_3 & \equiv \psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \vee \psi_2 \barwedge \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3 \\
\psi_1 \barwedge_{<} \phi \barwedge_{>} (\psi_2 \vee \psi_3) & \equiv \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \vee \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3 \\
\psi_1 \barwedge_{<} \phi \barwedge_{>} (\psi_2 \wedge \psi_3) & \equiv \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \barwedge \psi_3 \vee \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3 \barwedge \psi_2
\end{array}
$$

Figure 4.1: EOL equivalence rules.

Theorem 3. $\neg(\phi_1 \wedge \phi_2) \equiv \neg(\phi_1 \vee \phi_2)$

Proof. $\neg(\phi_1 \wedge \phi_2) \equiv \neg(\phi_1 \vee \phi_2)$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \Leftrightarrow \sigma \vDash_e \neg(\phi_1 \vee \phi_2)$.

$$
\begin{array}{rll}
\sigma[i..r] \vDash_e \neg(\phi_1 \wedge \phi_2) \Leftrightarrow & \sigma[i..r] \vDash_e \neg\phi_1 \text{ and } \sigma[i..r] \vDash_e \neg\phi_2 & \text{(Def. 21 (4.8))} \\
\Leftrightarrow & \sigma[i..r] \vDash_e \neg\phi_1 \wedge \neg\phi_2 & \text{(Def. 21 (4.6))} \\
\Leftrightarrow & \sigma[i..r] \vDash_e \neg(\phi_1 \vee \phi_2) & \text{(Theorem 4.2)}
\end{array}
$$

□

Theorem 4. $\psi_1 \wedge \psi_2 \equiv \psi_1 \barwedge \psi_2 \vee \psi_2 \barwedge \psi_1$

Proof. $\psi_1 \wedge \psi_2 \equiv \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_2 \mathbin{\barwedge} \psi_1$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \wedge \psi_2 \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_2 \mathbin{\barwedge} \psi_1$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \wedge \psi_2 \Leftrightarrow\ & \sigma[i..r] \vDash_e \psi_1 \text{ and } \sigma[i..r] \vDash_e \psi_2 && \text{(Def. 21 (4.6))} \\
\Leftrightarrow\ & \exists j : i \le j \le r \,.\, \sigma[i..j] \vDash_e \psi_1 \\
& \text{or } \sigma[j..r] \vDash_e \psi_1 && \text{(Def. 21 (4.3))} \\
& \text{and } \sigma[i..j] \vDash_e \psi_2 \\
& \text{or } \sigma[j..r] \vDash_e \psi_2 \\
\Leftrightarrow\ & \exists j : i \le j \le r \,.\, \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[j..r] \vDash_e \psi_2 \\
& \text{or } \sigma[i..j] \vDash_e \psi_2 \\
& \text{and } \sigma[j..r] \vDash_e \psi_1 \\
\Leftrightarrow\ & \sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_2 \mathbin{\barwedge} \psi_1 && \text{(Def. 21 (4.7) \& (4.12))}
\end{aligned}
$$

□

Theorem 5. $(\psi_1 \wedge \psi_2) \mathbin{\barwedge} \psi_3 \equiv \psi_1 \mathbin{\barwedge} \psi_3 \wedge \psi_2 \mathbin{\barwedge} \psi_3$

Proof. $(\psi_1 \wedge \psi_2) \mathbin{\barwedge} \psi_3 \equiv \psi_1 \mathbin{\barwedge} \psi_3 \wedge \psi_2 \mathbin{\barwedge} \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\psi_1 \wedge \psi_2) \mathbin{\barwedge} \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\barwedge} \psi_3 \wedge \psi_2 \mathbin{\barwedge} \psi_3$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e (\psi_1 \wedge \psi_2) \mathbin{\barwedge} \psi_3 \Leftrightarrow\ & \exists j,k : i \le j < k \le r \,. && \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e (\psi_1 \wedge \psi_2) \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j,k : i \le j < k \le r \,. \\
& \sigma[i..j] \vDash_e \psi_1 && \text{(Def. 21 (4.6))} \\
& \text{and } \sigma[i..j] \vDash_e \psi_2 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j,l,k : i \le j < k \le r \\
& \text{and } i \le l < k \le r \,. \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[i..l] \vDash_e \psi_2 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j,l,k : i \le j < k \le r \\
& \text{and } i \le l < k \le r \,. \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3
\end{aligned}
$$

$$
\begin{aligned}
& \text{and } \sigma[i..l] \vDash_e \psi_2 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} \psi_3 \wedge \psi_2 \mathbin{\barwedge} \psi_3 \qquad \text{(Def. 21 (4.6) \& (4.12))}
\end{aligned}
$$

□

Theorem 6. $\psi_1 \mathbin{\barwedge} (\psi_2 \wedge \psi_3) \equiv \psi_1 \mathbin{\barwedge} \psi_2 \wedge \psi_1 \mathbin{\barwedge} \psi_3$

Proof. $\psi_1 \mathbin{\barwedge} (\psi_2 \wedge \psi_3) \equiv \psi_1 \mathbin{\barwedge} \psi_2 \wedge \psi_1 \mathbin{\barwedge} \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \mathbin{\barwedge} (\psi_2 \wedge \psi_3) \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\barwedge} \psi_2 \wedge \psi_1 \mathbin{\barwedge} \psi_3$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} (\psi_2 \wedge \psi_3) \Leftrightarrow\ & \exists j,k : i \le j < k \le r\ . \qquad \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e (\psi_2 \wedge \psi_3) \\
\Leftrightarrow\ & \exists j,k : i \le j < k \le r\ . \qquad \text{(Def. 21 (4.6))} \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_2 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j,k : i \le j < k \le r\ . \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_2 \\
& \text{and } \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} \psi_2 \wedge \psi_1 \mathbin{\barwedge} \psi_3 \qquad \text{(Def. 21 (4.6) \& (4.12))}
\end{aligned}
$$

□

Theorem 7. $(\psi_1 \vee \psi_2) \mathbin{\barwedge} \psi_3 \equiv \psi_1 \mathbin{\barwedge} \psi_3 \vee \psi_2 \mathbin{\barwedge} \psi_3$

Proof. $(\psi_1 \vee \psi_2) \mathbin{\barwedge} \psi_3 \equiv \psi_1 \mathbin{\barwedge} \psi_3 \vee \psi_2 \mathbin{\barwedge} \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\psi_1 \vee \psi_2) \mathbin{\barwedge} \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\barwedge} \psi_3 \vee \psi_2 \mathbin{\barwedge} \psi_3$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e (\psi_1 \vee \psi_2) \mathbin{\barwedge} \psi_3 \Leftrightarrow\ & \exists j,k : i \le j < k \le r\ . \qquad \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e (\psi_1 \vee \psi_2) \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j,k : i \le j < k \le r\ . \qquad \text{(Def. 21 (4.7))} \\
& (\sigma[i..j] \vDash_e \psi_1 \\
& \text{or } \sigma[i..j] \vDash_e \psi_2)
\end{aligned}
$$

$$
\begin{aligned}
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j, l, k : i \le j < k \le r \\
& \text{and } i \le l < k \le r\ . \\
& (\sigma[i..j] \vDash_e \psi_1 \\
& \text{or } \sigma[i..l] \vDash_e \psi_2) \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \exists j, l, k : i \le j < k \le r \\
& \text{and } i \le l < k \le r\ . \\
& (\sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3) \\
& \text{or } (\sigma[i..l] \vDash_e \psi_2 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3) \\
\Leftrightarrow\ & \sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} \psi_3 \vee \psi_2 \mathbin{\barwedge} \psi_3 \quad \text{(Def. 21 (4.7) \& (4.12))}
\end{aligned}
$$

□

Theorem 8. $\psi_1 \mathbin{\barwedge} (\psi_2 \vee \psi_3) \equiv \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_1 \mathbin{\barwedge} \psi_3$

Proof. $\psi_1 \mathbin{\barwedge} (\psi_2 \vee \psi_3) \equiv \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_1 \mathbin{\barwedge} \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \mathbin{\barwedge} (\psi_2 \vee \psi_3) \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_1 \mathbin{\barwedge} \psi_3$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} (\psi_2 \vee \psi_3) \Leftrightarrow\ & \exists j, k : i \le j < k \le r\ . && \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e (\psi_2 \vee \psi_3) \\
\Leftrightarrow\ & \exists j, k : i \le j < k \le r\ . && \text{(Def. 21 (4.7))} \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } (\sigma[k..r] \vDash_e \psi_2 \\
& \text{or } \sigma[k..r] \vDash_e \psi_3) \\
\Leftrightarrow\ & \exists j, k : i \le j < k \le r\ . \\
& \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_2 \\
& \text{or } \sigma[i..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow\ & \sigma[i..r] \vDash_e \psi_1 \mathbin{\barwedge} \psi_2 \vee \psi_1 \mathbin{\barwedge} \psi_3 && \text{(Def. 21 (4.7) \& (4.12))}
\end{aligned}
$$

□

Theorem 9. $\psi_1 \mathbin{\barwedge} \psi_2 \wedge \psi_1 \mathbin{\barwedge} \psi_3 \equiv \psi_1 \mathbin{\barwedge} \psi_3 \wedge \psi_1 \mathbin{\barwedge} \psi_2$

Proof. $\psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \equiv \psi_1 \barwedge \psi_3 \wedge \psi_1 \barwedge \psi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge \psi_3 \wedge \psi_1 \barwedge \psi_2$.

$$
\begin{array}{lll}
\sigma[i..r] \vDash_e \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \Leftrightarrow & \sigma[i..r] \vDash_e \psi_1 \barwedge \psi_2 & \text{(Def. 21 (4.6))} \\
& \text{and } \sigma[i..r] \vDash_e \psi_1 \barwedge \psi_3 & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \psi_1 \barwedge \psi_3 \wedge \psi_1 \barwedge \psi_2 & \text{(Def. 21 (4.6))}
\end{array}
$$

□

Theorem 10. $\psi_1 \barwedge \psi_2 \barwedge \psi_3 \equiv \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \wedge \psi_2 \barwedge \psi_3$

Proof. $\psi_1 \barwedge \psi_2 \barwedge \psi_3 \equiv \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \wedge \psi_2 \barwedge \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge \psi_2 \barwedge \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \wedge \psi_2 \barwedge \psi_3$.

$$
\begin{array}{lll}
\sigma[i..r] \vDash_e \psi_1 \barwedge \psi_2 \barwedge \psi_3 \Leftrightarrow & \exists j,k,l : i \le j < k < l \le r\ . & \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e \psi_1 & \\
& \text{and } \sigma[k..l-1] \vDash_e \psi_2 & \\
& \text{and } \sigma[l..r] \vDash_e \psi_3 & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \psi_1 \barwedge \psi_2 \wedge \psi_1 \barwedge \psi_3 \wedge \psi_2 \barwedge \psi_3 & \text{(Def. 21 (4.6)} \\
& & \text{\& (4.12))}
\end{array}
$$

□

Theorem 11. $\psi_1 \barwedge_{[} (\phi_1 \wedge \phi_2) \equiv \psi_1 \barwedge_{[} \phi_1 \wedge \psi_1 \barwedge_{[} \phi_2$

Proof. $\psi_1 \barwedge_{[} (\phi_1 \wedge \phi_2) \equiv \psi_1 \barwedge_{[} \phi_1 \wedge \psi_1 \barwedge_{[} \phi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge_{[} (\phi_1 \wedge \phi_2) \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge_{[} \phi_1 \wedge \psi_1 \barwedge_{[} \phi_2$.

$$
\begin{array}{lll}
\sigma[i..r] \vDash_e \psi_1 \barwedge_{[} (\phi_1 \wedge \phi_2) \Leftrightarrow & (\exists j : i \le j \le r\ . & \text{(Def. 21 (4.13))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} & \\
& (\forall k : j \le k \le r\ . & \\
& \sigma[k..k] \vDash_e (\phi_1 \wedge \phi_2))) & \\
\Leftrightarrow & (\exists j : i \le j \le r\ . & \text{(Def. 21 (4.6))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} & \\
& (\forall k : j \le k \le r\ . & \\
& \sigma[k..k] \vDash_e \phi_1 & \\
& \text{and } \sigma[k..k] \vDash_e \phi_2)) & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \psi_1 \barwedge_{[} \phi_1 \wedge \psi_1 \barwedge_{[} \phi_2 & \text{(Def. 21 (4.13))}
\end{array}
$$

□

Theorem 12. $\psi_1 \mathbin{\mathbb{A}_{[}} (\phi_1 \vee \phi_2) \equiv \psi_1 \mathbin{\mathbb{A}_{[}} \phi_1 \vee \psi_1 \mathbin{\mathbb{A}_{[}} \phi_2$

Proof. $\psi_1 \mathbin{\mathbb{A}_{[}} (\phi_1 \vee \phi_2) \equiv \psi_1 \mathbin{\mathbb{A}_{[}} \phi_1 \vee \psi_1 \mathbin{\mathbb{A}_{[}} \phi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \mathbin{\mathbb{A}_{[}} (\phi_1 \vee \phi_2) \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\mathbb{A}_{[}} \phi_1 \vee \psi_1 \mathbin{\mathbb{A}_{[}} \phi_2$.

$$
\begin{array}{lll}
\sigma[i..r] \vDash_e \psi_1 \mathbin{\mathbb{A}_{[}} (\phi_1 \vee \phi_2) \Leftrightarrow & (\exists j : i \le j \le r \,. & \text{(Def. 21 (4.13))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} & \\
& (\forall k : j \le k \le r \,. & \\
& \sigma[k..k] \vDash_e (\phi_1 \vee \phi_2))) & \\
\Leftrightarrow & (\exists j : i \le j \le r \,. & \text{(Def. 21 (4.7))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} & \\
& (\forall k : j \le k \le r \,. & \\
& \sigma[k..k] \vDash_e \phi_1 & \\
& \text{ or } \sigma[k..k] \vDash_e \phi_2)) & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \psi_1 \mathbin{\mathbb{A}_{[}} \phi_1 \vee \psi_1 \mathbin{\mathbb{A}_{[}} \phi_2 & \text{(Def. 21 (4.13))}
\end{array}
$$

□

Theorem 13. $\psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_{[}} \phi \equiv \psi_1 \mathbin{\mathbb{A}} \psi_2 \wedge \psi_2 \mathbin{\mathbb{A}_{[}} \phi$

Proof. $\psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_{[}} \phi \equiv \psi_1 \mathbin{\mathbb{A}} \psi_2 \wedge \psi_2 \mathbin{\mathbb{A}_{[}} \phi$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_{[}} \phi \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\mathbb{A}} \psi_2 \wedge \psi_2 \mathbin{\mathbb{A}_{[}} \phi$.

$$
\begin{array}{lll}
\sigma[i..r] \vDash_e \psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_{[}} \phi \Leftrightarrow & \exists j,k : i \le j < k \le r \,. & \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e \psi_1 \text{ and} & \\
& \sigma[k..r] \vDash_e \psi_2 \mathbin{\mathbb{A}_{[}} \phi & \\
\Leftrightarrow & (\exists j,k : i \le j < k \le r \,. & \text{(Def. 21 (4.13))} \\
& \sigma[i..j] \vDash_e \psi_1 \text{ and} & \\
& \sigma[k..k] \vDash_e \psi_2 \text{ and} & \\
& (\forall l : k \le l \le r \,. & \\
& \sigma[l..l] \vDash_e \phi) & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \psi_1 \mathbin{\mathbb{A}} \psi_2 \wedge \psi_2 \mathbin{\mathbb{A}_{[}} \phi & \text{(Def. 21 (4.6), (4.12)} \\
& & \text{\& (4.13))}
\end{array}
$$

□

Theorem 14. $(\psi_1 \vee \psi_2) \mathbin{\mathbb{A}_{[}} \phi \equiv \psi_1 \mathbin{\mathbb{A}_{[}} \phi \vee \psi_2 \mathbin{\mathbb{A}_{[}} \phi$

Proof. $(\psi_1 \vee \psi_2) \mathbin{\mathbb{A}_{[}} \phi \equiv \psi_1 \mathbin{\mathbb{A}_{[}} \phi \vee \psi_2 \mathbin{\mathbb{A}_{[}} \phi$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\psi_1 \vee \psi_2) \mathbin{\mathbb{A}_{[}} \phi \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\mathbb{A}_{[}} \phi \vee \psi_2 \mathbin{\mathbb{A}_{[}} \phi$.

$$\sigma[i..r] \vDash_e (\psi_1 \vee \psi_2) \mathbin{\mathbb{A}_[} \phi \Leftrightarrow \quad \psi_1 \mathbin{\mathbb{A}_[} \phi \vee \psi_2 \mathbin{\mathbb{A}_[} \phi \qquad \text{(Theo. 7 \& Def. 21 (4.13))}$$

□

Theorem 15. $(\psi_1 \wedge \psi_2) \mathbin{\mathbb{A}_[} \phi \equiv \psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_[} \phi \vee \psi_2 \mathbin{\mathbb{A}} \psi_1 \mathbin{\mathbb{A}_[} \phi$

Proof. $(\psi_1 \wedge \psi_2) \mathbin{\mathbb{A}_[} \phi \equiv \psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_[} \phi \vee \psi_2 \mathbin{\mathbb{A}} \psi_1 \mathbin{\mathbb{A}_[} \phi$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\psi_1 \wedge \psi_2) \mathbin{\mathbb{A}_[} \phi \Leftrightarrow \sigma \vDash_e \psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_[} \phi \vee \psi_2 \mathbin{\mathbb{A}} \psi_1 \mathbin{\mathbb{A}_[} \phi$.

$$\begin{aligned}
\sigma[i..r] \vDash_e (\psi_1 \wedge \psi_2) \mathbin{\mathbb{A}_[} \phi \Leftrightarrow \quad & (\psi_1 \mathbin{\mathbb{A}} \psi_2 \vee \psi_2 \mathbin{\mathbb{A}} \psi_1) \mathbin{\mathbb{A}_[} \phi && \text{(Theo. 4)} \\
\Leftrightarrow \quad & \sigma[i..r] \vDash_e \psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}_[} \phi \vee \psi_2 \mathbin{\mathbb{A}} \psi_1 \mathbin{\mathbb{A}_[} \phi && \text{(Theo. 14)}
\end{aligned}$$

□

Theorem 16. $(\phi_1 \wedge \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \equiv \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \wedge \phi_2 \mathbin{\mathbb{A}_]} \psi_1$

Proof. $(\phi_1 \wedge \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \equiv \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \wedge \phi_2 \mathbin{\mathbb{A}_]} \psi_1$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\phi_1 \wedge \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \Leftrightarrow \sigma \vDash_e \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \wedge \phi_2 \mathbin{\mathbb{A}_]} \psi_1$.

$$\begin{aligned}
\sigma[i..r] \vDash_e (\phi_1 \wedge \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \Leftrightarrow \quad & (\exists j : i \le j \le r \,. && \text{(Def. 21 (4.14))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} \\
& (\forall k : 0 \le k \le j \,. \\
& \sigma[k..k] \vDash_e (\phi_1 \wedge \phi_2))) \\
\Leftrightarrow \quad & (\exists j : i \le j \le r \,. && \text{(Def. 21 (4.6))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} \\
& (\forall k : 0 \le k \le j \,. \\
& \sigma[k..k] \vDash_e \phi_1 \\
& \text{ and } \sigma[k..k] \vDash_e \phi_2)) \\
\Leftrightarrow \quad & \sigma[i..r] \vDash_e \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \wedge \phi_2 \mathbin{\mathbb{A}_]} \psi_1 && \text{(Def. 21 (4.14))}
\end{aligned}$$

□

Theorem 17. $(\phi_1 \vee \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \equiv \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \vee \phi_2 \mathbin{\mathbb{A}_]} \psi_1$

Proof. $(\phi_1 \vee \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \equiv \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \vee \phi_2 \mathbin{\mathbb{A}_]} \psi_1$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\phi_1 \vee \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \Leftrightarrow \sigma \vDash_e \phi_1 \mathbin{\mathbb{A}_]} \psi_1 \vee \phi_2 \mathbin{\mathbb{A}_]} \psi_1$.

$$\begin{aligned}
\sigma[i..r] \vDash_e (\phi_1 \vee \phi_2) \mathbin{\mathbb{A}_]} \psi_1 \Leftrightarrow \quad & (\exists j : i \le j \le r \,. && \text{(Def. 21 (4.14))} \\
& \sigma[j..j] \vDash_e \psi_1 \text{ and} \\
& (\forall k : 0 \le k \le j \,.
\end{aligned}$$

$$
\begin{array}{lll}
 & \sigma[k..k] \vDash_e (\phi_1 \vee \phi_2))) & \\
\Leftrightarrow & (\exists j : i \leq j \leq r \,. & \text{(Def. 21 (4.7))} \\
 & \sigma[j..j] \vDash_e \psi_1 \text{ and} & \\
 & (\forall k : 0 \leq k \leq j \,. & \\
 & \sigma[k..k] \vDash_e \phi_1 & \\
 & \text{ or } \sigma[k..k] \vDash_e \phi_2)) & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \phi_1 \mathbin{\text{A}_{]}} \psi_1 \vee \phi_2 \mathbin{\text{A}_{]}} \psi_1 & \text{(Def. 21 (4.14))}
\end{array}
$$

□

Theorem 18. $\phi \mathbin{\text{A}_{]}} \psi_1 \mathbin{\text{A}} \psi_2 \equiv \phi \mathbin{\text{A}_{]}} \psi_1 \wedge \psi_1 \mathbin{\text{A}} \psi_2$

Proof. $\phi \mathbin{\text{A}_{]}} \psi_1 \mathbin{\text{A}} \psi_2 \equiv \phi \mathbin{\text{A}_{]}} \psi_1 \wedge \psi_1 \mathbin{\text{A}} \psi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \mathbin{\text{A}} \psi_2 \Leftrightarrow \sigma \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \wedge \psi_1 \mathbin{\text{A}} \psi_2$.

$$
\begin{array}{lll}
\sigma[i..r] \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \mathbin{\text{A}} \psi_2 \Leftrightarrow & \exists j,k : i \leq j < k \leq r \,. & \text{(Def. 21 (4.12))} \\
 & \sigma[i..j] \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \text{ and} & \\
 & \sigma[k..r] \vDash_e \psi_2 & \\
\Leftrightarrow & (\exists j,k : i \leq j < k \leq r \,. & \text{(Def. 21 (4.13))} \\
 & \sigma[j..j] \vDash_e \psi_1 \text{ and} & \\
 & (\forall l : i \leq l \leq j \,. & \\
 & \sigma[l..l] \vDash_e \phi) \text{ and} & \\
 & \sigma[k..r] \vDash_e \psi_2 & \\
\Leftrightarrow & \sigma[i..r] \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \wedge \psi_1 \mathbin{\text{A}} \psi_2 & \text{(Def. 21 (4.6), (4.12)} \\
 & & \text{\& (4.13))}
\end{array}
$$

□

Theorem 19. $\phi \mathbin{\text{A}_{]}} (\psi_1 \vee \psi_2) \equiv \phi \mathbin{\text{A}_{]}} \psi_1 \vee \phi \mathbin{\text{A}_{]}} \psi_2$

Proof. $\phi \mathbin{\text{A}_{]}} (\psi_1 \vee \psi_2) \equiv \phi \mathbin{\text{A}_{]}} \psi_1 \vee \phi \mathbin{\text{A}_{]}} \psi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi \mathbin{\text{A}_{]}} (\psi_1 \vee \psi_2) \Leftrightarrow \sigma \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \vee \phi \mathbin{\text{A}_{]}} \psi_2$.

$$
\sigma[i..r] \vDash_e \phi \mathbin{\text{A}_{]}} (\psi_1 \vee \psi_2) \Leftrightarrow \quad \sigma \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \vee \phi \mathbin{\text{A}_{]}} \psi_2 \quad \text{(Theo. 8 \& Def. 21 (4.13))}
$$

□

Theorem 20. $\phi \mathbin{\text{A}_{]}} (\psi_1 \wedge \psi_2) \equiv \phi \mathbin{\text{A}_{]}} \psi_1 \mathbin{\text{A}} \psi_2 \vee \phi \mathbin{\text{A}_{]}} \psi_2 \mathbin{\text{A}} \psi_1$

Proof. $\phi \mathbin{\text{A}_{]}} (\psi_1 \wedge \psi_2) \equiv \psi_1 \mathbin{\text{A}} \psi_2 \mathbin{\text{A}_{[}} \phi \vee \psi_2 \mathbin{\text{A}} \psi_1 \mathbin{\text{A}_{[}} \phi$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi \mathbin{\text{A}_{]}} (\psi_1 \wedge \psi_2) \Leftrightarrow \sigma \vDash_e \phi \mathbin{\text{A}_{]}} \psi_1 \mathbin{\text{A}} \psi_2 \vee \phi \mathbin{\text{A}_{]}} \psi_2 \mathbin{\text{A}} \psi_1$.

$$\begin{aligned}
\sigma[i..r] \vDash_e \phi \,\barwedge_{]}\, (\psi_1 \wedge \psi_2) \Leftrightarrow\ & \phi \,\barwedge_{]}\, (\psi_1 \barwedge \psi_2 \vee \psi_2 \barwedge \psi_1) && \text{(Theo. 4)}\\
\Leftrightarrow\ & \sigma[i..r] \vDash_e \phi \,\barwedge_{]}\, \psi_1 \barwedge \psi_2 \vee \phi \,\barwedge_{]}\, \psi_2 \barwedge \psi_1 && \text{(Theo. 19)}
\end{aligned}$$

□

Theorem 21. $\psi_1 \barwedge_{<} (\phi_1 \vee \phi_2) \barwedge_{>} \psi_2 \equiv \psi_1 \barwedge_{<} \phi_1 \barwedge_{>} \psi_2 \vee \psi_1 \barwedge_{<} \phi_2 \barwedge_{>} \psi_2$

Proof. $\psi_1 \barwedge_{<} (\phi_1 \vee \phi_2) \barwedge_{>} \psi_2 \equiv \psi_1 \barwedge_{<} \phi_1 \barwedge_{>} \psi_2 \vee \psi_1 \barwedge_{<} \phi_2 \barwedge_{>} \psi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge_{<} (\phi_1 \vee \phi_2) \barwedge_{>} \psi_2 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge_{<} \phi_1 \barwedge_{>} \psi_2 \vee \psi_1 \barwedge_{<} \phi_2 \barwedge_{>} \psi_2$.

$$\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \barwedge_{<} (\phi_1 \vee \phi_2) \barwedge_{>} \psi_2 \Leftrightarrow\ & (\exists j,k : i \le j < k \le r\ . && \text{(Def. 21 (4.15))}\\
& \sigma[j..j] \vDash_e \psi_1\\
& \text{ and } \sigma[k..k] \vDash_e \psi_2\\
& \text{ and } (\forall l : j \le l \le k\ .\\
& \sigma[l..l] \vDash_e (\phi_1 \vee \phi_2)))\\
\Leftrightarrow\ & (\exists j,k : i \le j < k \le r\ . && \text{(Def. 21 (4.7))}\\
& \sigma[j..j] \vDash_e \psi_1\\
& \text{ and } \sigma[k..k] \vDash_e \psi_2\\
& \text{ and } (\forall l : j \le l \le k\ .\\
& \sigma[l..l] \vDash_e \phi_1\\
& \text{ or } \sigma[l..l] \vDash_e \phi_2))\\
\Leftrightarrow\ & \psi_1 \barwedge_{<} \phi_1 \barwedge_{>} \psi_2\\
& \vee \psi_1 \barwedge_{<} \phi_2 \barwedge_{>} \psi_2 && \text{(Def. 21 (4.15))}
\end{aligned}$$

□

Theorem 22. $\psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \equiv \psi_1 \barwedge \psi_2 \wedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3$

Proof. $\psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \equiv \psi_1 \barwedge \psi_2 \wedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge \psi_2 \wedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3$.

$$\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3 \Leftrightarrow\ & \exists j,k : i \le j < k \le r\ . && \text{(Def. 21 (4.12))}\\
& \sigma[i..j] \vDash_e \psi_1 \text{ and}\\
& \sigma[k..r] \vDash_e \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3\\
\Leftrightarrow\ & (\exists j,k,l : i \le j < k < l \le r\ . && \text{(Def. 21 (4.15))}\\
& \sigma[i..j] \vDash_e \psi_1\\
& \text{ and } \sigma[k..k] \vDash_e \psi_2\\
& \text{ and } \sigma[l..l] \vDash_e \psi_3
\end{aligned}$$

$$
\begin{aligned}
& \text{and } (\forall m : k \le m \le l \,. \\
& \sigma[m..m] \vDash_e \phi) \\
\Leftrightarrow \quad & \psi_1 \barwedge \psi_2 \wedge \psi_2 \barwedge_< \phi \barwedge_> \psi_3 \qquad \text{(Def. 21 (4.6),} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(4.12) \& (4.15))}
\end{aligned}
$$

□

Theorem 23. $\psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \wedge \psi_2 \barwedge \psi_3$

Proof. $\psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \wedge \psi_2 \barwedge \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \wedge \psi_2 \barwedge \psi_3$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \Leftrightarrow \quad & \exists j,k : i \le j < k \le r \,. \qquad \text{(Def. 21 (4.12))} \\
& \sigma[i..j] \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \text{ and} \\
& \sigma[k..r] \vDash_e \psi_3 \\
\Leftrightarrow \quad & (\exists j,k,l : i \le j < k < l \le r \,. \qquad \text{(Def. 21 (4.15))} \\
& \sigma[j..j] \vDash_e \psi_1 \\
& \text{and } \sigma[k..k] \vDash_e \psi_2 \\
& \text{and } (\forall m : j \le m \le k \,. \\
& \sigma[m..m] \vDash_e \phi) \\
& \text{and } \sigma[l..r] \vDash_e \psi_3 \\
\Leftrightarrow \quad & \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \wedge \psi_2 \barwedge \psi_3 \qquad \text{(Def. 21 (4.6),} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(4.12) \& (4.15))}
\end{aligned}
$$

□

Theorem 24. $(\psi_1 \vee \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge_< \phi \barwedge_> \psi_3$

Proof. $(\psi_1 \vee \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge_< \phi \barwedge_> \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\psi_1 \vee \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge_< \phi \barwedge_> \psi_3$.

$$
\begin{aligned}
\sigma[i..r] \vDash_e (\psi_1 \vee \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \Leftrightarrow \quad & \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge_< \phi \barwedge_> \psi_3 \qquad \text{(Theo. 7 \&} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Def. 21 (4.15))}
\end{aligned}
$$

□

Theorem 25. $(\psi_1 \wedge \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \equiv \psi_1 \barwedge \psi_2 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge \psi_1 \barwedge_< \phi \barwedge_> \psi_3$

Proof. $(\psi_1 \wedge \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \equiv \psi_1 \barwedge \psi_2 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge \psi_1 \barwedge_< \phi \barwedge_> \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e (\psi_1 \wedge \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge \psi_2 \barwedge_< \phi \barwedge_> \psi_3 \vee \psi_2 \barwedge \psi_1 \barwedge_< \phi \barwedge_> \psi_3$.

$$\begin{aligned}
\sigma[i..r] \vDash_e (\psi_1 \wedge \psi_2) \barwedge_< \phi \barwedge_> \psi_3 \Leftrightarrow \quad & (\psi_1 \barwedge \psi_2 \vee \psi_2 \barwedge \psi_1) \barwedge_< \phi \barwedge_> \psi_3 && \text{(Theo. 4)} \\
\Leftrightarrow \quad & \psi_1 \barwedge \psi_2 \barwedge_< \phi \barwedge_> \psi_3 \vee && \text{(Theo. 24)} \\
& \psi_2 \barwedge \psi_1 \barwedge_< \phi \barwedge_> \psi_3
\end{aligned}$$

□

Theorem 26. $\psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \vee \psi_3) \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3$

Proof. $\psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \vee \psi_3) \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \vee \psi_3) \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3$.

$$\sigma[i..r] \vDash_e \psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \vee \psi_3) \Leftrightarrow \quad \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \quad \text{(Theo. 8 \& Def. 21 (4.15))}$$

□

Theorem 27. $\psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \wedge \psi_3) \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \barwedge \psi_2$

Proof. $\psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \wedge \psi_3) \equiv \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \barwedge \psi_2$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \wedge \psi_3) \Leftrightarrow \sigma \vDash_e \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \vee \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \barwedge \psi_2$.

$$\begin{aligned}
\sigma[i..r] \vDash_e \psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \wedge \psi_3) \Leftrightarrow \quad & \psi_1 \barwedge_< \phi \barwedge_> (\psi_2 \barwedge \psi_3 \vee \psi_3 \barwedge \psi_2) && \text{(Theo. 4)} \\
\Leftrightarrow \quad & \psi_1 \barwedge_< \phi \barwedge_> \psi_2 \barwedge \psi_3 \vee && \text{(Theo. 26)} \\
& \psi_1 \barwedge_< \phi \barwedge_> \psi_3 \barwedge \psi_2
\end{aligned}$$

□

In order to be able to define translation rules from EOL to LTL it is necessary to define a normal form for EOL formulas, that we refer to as event order normal form (EONF). The EONF prohibits the unordered *and*- ($\wedge$) and *or*-operator ($\vee$) to appear in the operands of any ordered operator but permits the *and*-operator ($\wedge$) if it appears in an operand of the between operators $\barwedge_<$ and $\barwedge_>$.

Definition 25. *Event Order Normal Form (EONF). The set of EOL formulas over a set $\mathcal{A}$ of event variables in event order normal form (EONF) is given by:*

$$\phi ::= a \mid \neg\phi$$

$$\phi_\wedge ::= \phi \mid \phi_\wedge \mid \neg\phi_\wedge \mid \phi_{\wedge_1} \wedge \phi_{\wedge_2}$$

$$\psi ::= \phi \mid \phi_1 \barwedge \phi_2 \mid \phi \barwedge_[\phi \mid \phi \barwedge_] \phi \mid \phi_1 \barwedge_< \phi_\wedge \barwedge_> \phi_2$$

$$\psi_\wedge ::= \psi \mid \psi_\wedge \mid \psi_{\wedge_1} \wedge \psi_{\wedge_2} \mid \psi_{\wedge_1} \vee \psi_{\wedge_2}$$

where $a \in \mathcal{A}$ and ϕ, ϕ_1, ϕ_2 are simple EOL formulas in EONF and $\phi_\wedge$, $\phi_{\wedge_1}$ and $\phi_{\wedge_2}$ are simple EOL formulas containing the $\wedge$-operator in EONF and $\psi_\wedge$, $\psi_{\wedge_1}$ and $\psi_{\wedge_2}$ are complex EOL formulas containing the $\wedge$-operator and / or $\vee$-operator in EONF.

An EOL formula can be transformed into an equivalent EOL formula in EONF by rewriting using the equivalence rules from Figure 4.1. For instance, the EOL formula $\psi = \mathrm{Ta} \mathbin{\mathbb{A}} \mathrm{Gc} \mathbin{\mathbb{A}} \mathrm{Tc}$ can be rewritten in EONF as $\psi' = (\mathrm{Ta} \mathbin{\mathbb{A}} \mathrm{Gc}) \wedge (\mathrm{Gc} \mathbin{\mathbb{A}} \mathrm{Tc}) \wedge (\mathrm{Ta} \mathbin{\mathbb{A}} \mathrm{Tc})$.

Theorem 28. *For each EOL formula there exists a semantically equivalent EOL formula in EONF.*

Proof. The following rewrite rules can be applied recursively to the syntactic structure of a EOL formula that is not in EONF and will create a semantically equivalent EOL formula which is in EONF.

$$\begin{aligned}
\mathrm{EONF}(a) &= a \\
\mathrm{EONF}(\neg a) &= \neg \mathrm{EONF}(a) \\
\mathrm{EONF}(\phi_1 \wedge \phi_2) &= \mathrm{EONF}(\phi_1) \wedge \mathrm{EONF}(\phi_2) \\
\mathrm{EONF}(\phi_1 \vee \phi_2) &= \mathrm{EONF}(\phi_1) \vee \mathrm{EONF}(\phi_2) \\
\mathrm{EONF}(\neg(\phi_1 \wedge \phi_2)) &= \mathrm{EONF}(\neg\phi_1) \wedge \mathrm{EONF}(\neg\phi_2) \\
\mathrm{EONF}(\neg(\phi_1 \vee \phi_2)) &= \mathrm{EONF}(\neg\phi_1) \wedge \mathrm{EONF}(\neg\phi_2) \\
\mathrm{EONF}(a_1 \mathbin{\mathbb{A}} a_2) &= \mathrm{EONF}(a_1) \mathbin{\mathbb{A}} \mathrm{EONF}(a_2) \\
\mathrm{EONF}(a_1 \mathbin{\mathbb{A}}_[a_2) &= \mathrm{EONF}(a_1) \mathbin{\mathbb{A}}_[\mathrm{EONF}(a_2) \\
\mathrm{EONF}(a_1 \mathbin{\mathbb{A}}_] a_2) &= \mathrm{EONF}(a_1) \mathbin{\mathbb{A}}_] \mathrm{EONF}(a_2) \\
\mathrm{EONF}(a_1 \mathbin{\mathbb{A}}_< a_2 \mathbin{\mathbb{A}}_> a_3) &= \mathrm{EONF}(a_1) \mathbin{\mathbb{A}}_< \mathrm{EONF}(a_2) \mathbin{\mathbb{A}}_> \mathrm{EONF}(a_3) \\
\mathrm{EONF}(\psi_1 \wedge \psi_2) &= \mathrm{EONF}(\psi_1) \wedge \mathrm{EONF}(\psi_2) \\
\mathrm{EONF}(\psi_1 \vee \psi_2) &= \mathrm{EONF}(\psi_1) \vee \mathrm{EONF}(\psi_2) \\
\mathrm{EONF}((\psi_1 \wedge \psi_2) \mathbin{\mathbb{A}} \psi_3) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_3) \wedge \mathrm{EONF}(\psi_2 \mathbin{\mathbb{A}} \psi_3) \\
\mathrm{EONF}((\psi_1 \vee \psi_2) \mathbin{\mathbb{A}} \psi_3) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_3) \vee \mathrm{EONF}(\psi_2 \mathbin{\mathbb{A}} \psi_3) \\
\mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} (\psi_2 \wedge \psi_3)) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2) \wedge \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_3) \\
\mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} (\psi_2 \vee \psi_3)) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2) \vee \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_3) \\
\mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}} \psi_3) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2) \wedge \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_3) \wedge \\
&\quad \mathrm{EONF}(\psi_2 \mathbin{\mathbb{A}} \psi_3) \\
\mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}}_[\phi) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2) \wedge \mathrm{EONF}(\psi_2 \mathbin{\mathbb{A}}_[\phi) \\
\mathrm{EONF}((\psi_1 \vee \psi_2) \mathbin{\mathbb{A}}_[\phi) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}}_[\phi) \vee \mathrm{EONF}(\psi_2 \mathbin{\mathbb{A}}_[\phi) \\
\mathrm{EONF}((\psi_1 \wedge \psi_2) \mathbin{\mathbb{A}}_[\phi) &= \mathrm{EONF}(\psi_1 \mathbin{\mathbb{A}} \psi_2 \mathbin{\mathbb{A}}_[\phi) \vee \mathrm{EONF}(\psi_2 \mathbin{\mathbb{A}} \psi_1 \mathbin{\mathbb{A}}_[\phi) \\
\mathrm{EONF}(\psi \mathbin{\mathbb{A}}_[(\phi_1 \wedge \phi_2)) &= \mathrm{EONF}(\psi \mathbin{\mathbb{A}}_[\phi_1) \wedge \mathrm{EONF}(\psi \mathbin{\mathbb{A}}_[\phi_2) \\
\mathrm{EONF}(\psi \mathbin{\mathbb{A}}_[(\phi_1 \vee \phi_2)) &= \mathrm{EONF}(\psi \mathbin{\mathbb{A}}_[\phi_1) \vee \mathrm{EONF}(\psi \mathbin{\mathbb{A}}_[\phi_2)
\end{aligned}$$

$$
\begin{aligned}
\mathrm{EONF}(\psi \barwedge_{[} \neg(\phi_1 \wedge \phi_2)) &= \mathrm{EONF}(\psi \barwedge_{[} (\neg\phi_1 \wedge \neg\phi_2))\\
\mathrm{EONF}(\psi \barwedge_{[} \neg(\phi_1 \vee \phi_2)) &= \mathrm{EONF}(\psi \barwedge_{[} \neg(\phi_1 \wedge \phi_2))\\
\mathrm{EONF}(\phi \barwedge_{]} \psi_1 \barwedge \psi_2) &= \mathrm{EONF}(\phi \barwedge_{]} \psi_1) \wedge \mathrm{EONF}(\psi_1 \barwedge \psi_2)\\
\mathrm{EONF}(\phi \barwedge_{]} (\psi_1 \vee \psi_2)) &= \mathrm{EONF}(\phi \barwedge_{]} \psi_1) \vee \mathrm{EONF}(\phi \barwedge_{]} \psi_2)\\
\mathrm{EONF}(\phi \barwedge_{]} (\psi_1 \wedge \psi_2)) &= \mathrm{EONF}(\phi \barwedge_{]} \psi_1 \barwedge \psi_2) \vee \mathrm{EONF}(\phi \barwedge_{]} \psi_2 \barwedge \psi_1)\\
\mathrm{EONF}((\phi_1 \wedge \phi_2) \barwedge_{]} \psi) &= \mathrm{EONF}(\phi_1 \barwedge_{]} \psi) \wedge \mathrm{EONF}(\phi_2 \barwedge_{]} \psi)\\
\mathrm{EONF}((\phi_1 \vee \phi_2) \barwedge_{]} \psi) &= \mathrm{EONF}(\phi_1 \barwedge_{]} \psi) \vee \mathrm{EONF}(\phi_2 \barwedge_{]} \psi_3)\\
\mathrm{EONF}(\neg(\phi_1 \wedge \phi_2) \barwedge_{]} \psi) &= \mathrm{EONF}((\neg\phi_1 \wedge \neg\phi_2) \barwedge_{]} \psi)\\
\mathrm{EONF}(\neg(\phi_1 \vee \phi_2) \barwedge_{]} \psi) &= \mathrm{EONF}(\neg(\phi_1 \wedge \phi_2) \barwedge_{]} \psi)\\
\mathrm{EONF}(\psi_1 \barwedge_{<} (\phi_1 \vee \phi_2) \barwedge_{>} \psi_2) &= \mathrm{EONF}(\psi_1 \barwedge_{<} \phi_1 \barwedge_{>} \psi_2) \vee \mathrm{EONF}(\psi_1 \barwedge_{<} \phi_2 \barwedge_{>} \psi_2)\\
\mathrm{EONF}(\psi_1 \barwedge_{<} \neg(\phi_1 \vee \phi_2) \barwedge_{>} \psi_2) &= \mathrm{EONF}(\psi_1 \barwedge_{<} \neg(\phi_1 \wedge \phi_2) \barwedge_{>} \psi_2)\\
\mathrm{EONF}(\psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3) &= \mathrm{EONF}(\psi_1 \barwedge \psi_2) \wedge \mathrm{EONF}(\psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3)\\
\mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \barwedge \psi_3) &= \mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2) \wedge \mathrm{EONF}(\psi_2 \barwedge \psi_3)\\
\mathrm{EONF}((\psi_1 \wedge \psi_2) \barwedge_{<} \phi \barwedge_{>} \psi_3) &= \mathrm{EONF}(\psi_1 \barwedge \psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3) \vee\\
&\quad \mathrm{EONF}(\psi_2 \barwedge \psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3)\\
\mathrm{EONF}((\psi_1 \vee \psi_2) \barwedge_{<} \phi \barwedge_{>} \psi_3) &= \mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3) \vee \mathrm{EONF}(\psi_2 \barwedge_{<} \phi \barwedge_{>} \psi_3)\\
\mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} (\psi_2 \wedge \psi_3)) &= \mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2 \barwedge \psi_3) \vee\\
&\quad \mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3 \barwedge \psi_2)\\
\mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} (\psi_2 \vee \psi_3)) &= \mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2) \vee \mathrm{EONF}(\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_3)
\end{aligned}
$$

The correctness of the transformations and corresponding equivalences have been shown in Theorems 1-27. For every syntactically valid EOL formula there exists an semantically equivalent rewriting in EONF that can be achieved by the above rewriting rules.

□

4.3 Relationship to Linear Temporal Logic

In this section we will show that it is possible to translate each EOL formula into an equivalent LTL formula. An EOL formula and an LTL formula are equivalent if they are satisfied by the same set of execution traces.

Definition 26. *Equivalence of an EOL and an LTL formula. An EOL formula ψ and an LTL formula φ are equivalent denoted by $\psi \equiv \varphi$ if any for transition system T and all traces σ in T: $\sigma \vDash_e \psi \Leftrightarrow \sigma \vDash_l \varphi$.*

We will now define translation rules that can be used to translate a EOL formula in EONF into a equivalent LTL formula. The translation rules are based on the LTL specification patterns proposed by Dwyer et al. in [38].

Definition 27. *LTL formula for an EOL formula. Let ψ an EOL formula that is built over the set of event variables $a \in \mathcal{A}$ and which is in EONF. The states in*

T are labeled with an atomic proposition indicating whether the event represented by the event variable a leads to this state. More formally, if $s_{j-1} \xrightarrow{\alpha_i} s_j$ *then* $a_{\alpha_i} \in L(s_j)$*. The equivalent EOL formula for an EOL formula* ψ *can be constructed as follows:*

If ψ *does contain one of the ordered operators* $\wedge$, $\wedge_[$, $\wedge_]$, *or* $\wedge_<...\wedge_>$ *the translation function* $LTL_\wedge(\psi)$ *is used, otherwise,* $LTL(\psi)$ *is used. The translation functions* $LTL_\wedge(\psi)$ *and* $LTL(\psi)$ *are applied recursively to the syntactic structure of* ψ*.*

$$
\begin{aligned}
LTL(a_{\alpha_i}) &= \Diamond a_{\alpha_i} \\
LTL(\neg a_{\alpha_i}) &= \Box \neg a_{\alpha_i} \\
LTL(\phi_1 \wedge \phi_2) &= LTL(\phi_1) \wedge LTL(\phi_2) \\
LTL(\neg(\phi_1 \wedge \phi_2)) &= LTL(\neg\phi_1) \wedge LTL(\neg\phi_2) \\
LTL_\wedge(a_{\alpha_i}) &= a_{\alpha_i} \\
LTL_\wedge(\neg a_{\alpha_i}) &= \neg a_{\alpha_i} \\
LTL_\wedge(\phi_1 \wedge \phi_2) &= LTL_\wedge(\phi_1) \wedge LTL_\wedge(\phi_2) \\
LTL_\wedge(\neg(\phi_1 \wedge \phi_2)) &= LTL_\wedge(\neg\phi_1) \wedge LTL_\wedge(\neg\phi_2) \\
LTL_\wedge(\psi_{\wedge_1} \wedge \psi_{\wedge_2}) &= LTL_\wedge(\psi_{\wedge_1}) \wedge LTL_\wedge(\psi_{\wedge_2}) \\
LTL_\wedge(\psi_{\wedge_1} \vee \psi_{\wedge_2}) &= LTL_\wedge(\psi_{\wedge_1}) \vee LTL_\wedge(\psi_{\wedge_2}) \\
LTL_\wedge(\phi_1 \wedge \phi_2) &= \Diamond(LTL_\wedge(\phi_1) \wedge \Diamond LTL_\wedge(\phi_2)) \\
LTL_\wedge(\phi_1 \wedge_[\phi_2) &= \Diamond(LTL_\wedge(\phi_1) \wedge \Box LTL_\wedge(\phi_2)) \\
LTL_\wedge(\phi_1 \wedge_] \phi_2) &= LTL_\wedge(\phi_1)\, \mathcal{U}\, LTL_\wedge(\phi_2) \\
LTL_\wedge(\phi_1 \wedge_< \phi_\wedge \wedge_> \phi_2) &= \Diamond(LTL_\wedge(\phi_1) \wedge (LTL_\wedge(\phi_\wedge)\, \mathcal{U}\, LTL_\wedge(\phi_2)))
\end{aligned}
$$

where a_{α_i} *is an event variable and the corresponding atomic proposition with which the state is labeled,* ϕ, ϕ_1, ϕ_2 *are simple EOL formulas in EONF,* $\phi_\wedge$ *is a simple EOL formula in EONF containing the* $\wedge$*-operator, and* $\psi_\wedge$, $\psi_{\wedge_1}$ *and* $\psi_{\wedge_2}$ *are complex EOL formulas containing the* $\wedge$*-operator and / or* $\vee$*-operator in EONF.*

The translation rules define a translation from EOL formulas to LTL formulas on a syntactic level. It remains to be shown that the translation rules defined in Definition 27 translate an EOL formula in an semantically equivalent LTL formula with respect to Definition 26. Consequently, we show in Theorem 29 to Theorem 42 that all traces that are accepted by an EOL formula are also accepted by the corresponding LTL formula obtained by applying the translation rules.

Theorem 29. $s_j \vDash_e a_{\alpha_i} \equiv s_j \vDash_l a_{\alpha_i}$

Proof. $s_j \vDash_e a_{\alpha_i} \equiv s_j \vDash_l a_{\alpha_i}$ holds if for any transition system T and all states s in T: $s \vDash_e a_{\alpha_i} \Leftrightarrow s \vDash_l a_{\alpha_i}$.

$$
\begin{aligned}
s_j \vDash_e a_{\alpha_i} \quad &\Leftrightarrow \quad s_j \vDash_l a_{\alpha_i} \\
s_j \vDash_e a_{\alpha_i} \textit{ iff } s_{j-1} \xrightarrow{\alpha_i} s_j \quad &\Leftrightarrow \quad s_j \vDash_l a_{\alpha_i} \text{ iff } a_{\alpha_i} \in L(s_j)
\end{aligned}
$$

$$\text{Per definition } a_{\alpha_i} \in L(s_j) \text{ holds if } s_{j-1} \xrightarrow{\alpha_i} s_j.$$

□

Theorem 30. $s_j \vDash_e \neg a_{\alpha_i} \equiv s_j \vDash_l \neg a_{\alpha_i}$

Proof. $s_j \vDash_e \neg a_{\alpha_i} \equiv s_j \vDash_l \neg a_{\alpha_i}$ holds if for any transition system T and all states s in T: $s \vDash_e \neg a_{\alpha_i} \Leftrightarrow s \vDash_l \neg a_{\alpha_i}$.

$$s_j \vDash_e \neg a_{\alpha_i} \quad \Leftrightarrow \quad s_j \vDash_l \neg a_{\alpha_i}$$

$$s_j \vDash_e \neg a_{\alpha_i} \textit{ iff not } \; s_{j-1} \xrightarrow{\alpha_i} s_j \quad \Leftrightarrow \quad s_j \vDash_l \neg a_{\alpha_i} \text{ iff not } a_{\alpha_i} \in L(s_j)$$

□

Theorem 31. $\sigma \vDash_e a_{\alpha_i} \equiv \sigma \vDash_l \Diamond a_{\alpha_i}$

Proof. $\sigma \vDash_e a_{\alpha_i} \equiv \sigma \vDash_l \Diamond a_{\alpha_i}$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e a_{\alpha_i} \Leftrightarrow \sigma \vDash_l \Diamond a_{\alpha_i}$.

$$\sigma \vDash_e a_{\alpha_i} \quad \Leftrightarrow \quad \sigma \vDash_l \Diamond a_{\alpha_i}$$

$$\sigma \vDash_e a_{\alpha_i} \textit{ iff } \exists j : 0 \le j \le n \,.\, s_j \vDash_e a_{\alpha_i} \quad \Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \text{ true } \mathcal{U}\, a_{\alpha_i} \text{ iff } \exists k \ge 0 \,.\, \sigma[k...] \vDash_l a_{\alpha_i} \\ \text{and } \forall j : 0 \le j < k \,.\, \sigma[j...] \vDash_l \text{ true} \end{array}$$

$$\Leftrightarrow \quad \sigma \vDash_l \text{ true } \mathcal{U}\, a_{\alpha_i} \text{ iff } \exists k \ge 0 \,.\, \sigma[k...] \vDash_l a_{\alpha_i}$$

□

Theorem 32. $\sigma \vDash_e \neg a_{\alpha_i} \equiv \sigma \vDash_l \Box\neg a_{\alpha_i}$

Proof. $\sigma \vDash_e \neg a_{\alpha_i} \equiv \sigma \vDash_l \Box\neg a_{\alpha_i}$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \neg a_{\alpha_i} \Leftrightarrow \sigma \vDash_l \Box\neg a_{\alpha_i}$.

$$\sigma \vDash_e \neg a_{\alpha_i} \quad \Leftrightarrow \quad \sigma \vDash_l \Box\neg a_{\alpha_i}$$

$$\begin{array}{c} \sigma \vDash_e \neg a_{\alpha_i} \textit{ iff} \\ \forall j : 0 \le j \le n \,.\, s_j \vDash_e \neg a_{\alpha_i} \end{array} \quad \Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \neg(\text{ true } \mathcal{U}\, \neg\neg a_{\alpha_i}) \text{ iff not} \\ \exists k \ge 0 \,.\, \sigma[k...] \vDash_l \neg\neg a_{\alpha_i} \end{array}$$

$$\Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \neg(\text{ true } \mathcal{U}\, a_{\alpha_i}) \text{ iff not} \\ \exists k \ge 0 \,.\, \sigma[k...] \vDash_l a_{\alpha_i} \end{array}$$

$$\Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \neg(\text{ true } \mathcal{U}\, a_{\alpha_i}) \text{ iff} \\ \forall k \ge 0 \,.\, \sigma[k...] \vDash_l \neg a_{\alpha_i} \end{array}$$

□

Theorem 33. $\sigma \vDash_e \phi_1 \wedge \phi_2 \equiv \sigma \vDash_l LTL(\phi_1) \wedge LTL(\phi_2)$

Proof. $\sigma \vDash_e \phi_1 \wedge \phi_2 \equiv \sigma \vDash_l \mathrm{LTL}(\phi_1) \wedge \mathrm{LTL}(\phi_2)$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi_1 \wedge \phi_2 \Leftrightarrow \sigma \vDash_l \mathrm{LTL}(\phi_1) \wedge \mathrm{LTL}(\phi_2)$

$$\sigma \vDash_e \phi_1 \wedge \phi_2 \quad \Leftrightarrow \quad \sigma \vDash_l \mathrm{LTL}(\phi_1) \wedge \mathrm{LTL}(\phi_2)$$

$$\begin{array}{c} \sigma \vDash_e \phi_1 \wedge \phi_2 \textit{ iff} \\ \sigma \vDash_e \phi_1 \text{ and } \sigma \vDash_e \phi_2 \end{array} \Leftrightarrow \begin{array}{c} \sigma \vDash_l \mathrm{LTL}(\phi_1) \wedge \mathrm{LTL}(\phi_2) \text{ iff} \\ \sigma \vDash_l \mathrm{LTL}(\phi_1) \text{ and } \sigma \vDash_l \mathrm{LTL}(\phi_2) \end{array}$$

□

Theorem 34. $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \equiv \sigma \vDash_l \mathit{LTL}(\neg\phi_1) \wedge \mathit{LTL}(\neg\phi_2)$

Proof. $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \equiv \sigma \vDash_l \mathrm{LTL}(\neg\phi_1) \wedge \mathrm{LTL}(\neg\phi_2)$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \Leftrightarrow \sigma \vDash_l \mathrm{LTL}(\neg\phi_1) \wedge \mathrm{LTL}(\neg\phi_2)$

$$\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \quad \Leftrightarrow \quad \sigma \vDash_l \mathrm{LTL}(\neg\phi_1) \wedge \mathrm{LTL}(\neg\phi_2)$$

$$\begin{array}{c} \sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \textit{ iff} \\ \sigma \vDash_e \neg\phi_1 \text{ and } \sigma \vDash_e \neg\phi_2 \end{array} \Leftrightarrow \begin{array}{c} \sigma \vDash_l \mathrm{LTL}(\neg\phi_1) \wedge \mathrm{LTL}(\neg\phi_2) \text{ iff} \\ \sigma \vDash_l \mathrm{LTL}(\neg\phi_1) \text{ and } \mathrm{LTL}(\neg\phi_2) \end{array}$$

□

Theorem 35. $\sigma \vDash_e \phi_1 \wedge \phi_2 \equiv \sigma \vDash_l \mathit{LTL}_{\barwedge}(\phi_1) \wedge \mathit{LTL}_{\barwedge}(\phi_2)$

Proof. $\sigma \vDash_e \phi_1 \wedge \phi_2 \equiv \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\phi_2)$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi_1 \wedge \phi_2 \Leftrightarrow \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\phi_2)$. The translation rule $\mathrm{LTL}_{\barwedge}$ is only applied to $\phi_1 \wedge \phi_2$ if $\phi_1 \wedge \phi_2$ occurs in the formula $\phi_{1'} \barwedge_{<} (\phi_1 \wedge \phi_2) \barwedge_{>} \phi_{2'}$ and, consequently, according to the semantics of EOL $\phi_1 \wedge \phi_2$ has to hold on some $\sigma[l..l] \equiv s_l$.

$$s_l \vDash_e \phi_1 \wedge \phi_2 \quad \Leftrightarrow \quad s_l \vDash_l \mathrm{LTL}_{\barwedge}(\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\phi_2)$$

$$\begin{array}{c} s_l \vDash_e \phi_1 \wedge \phi_2 \textit{ iff} \\ s_l \vDash_e \phi_1 \text{ and } s_l \vDash_e \phi_2 \end{array} \Leftrightarrow \begin{array}{c} s_l \vDash_l \mathrm{LTL}_{\barwedge}(\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\phi_2) \text{ iff} \\ s_l \vDash_l \mathrm{LTL}_{\barwedge}(\phi_1) \text{ and } s_l \vDash_l \mathrm{LTL}_{\barwedge}(\phi_2) \end{array}$$

□

Theorem 36. $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \equiv \sigma \vDash_l \mathit{LTL}_{\barwedge}(\neg\phi_1) \wedge \mathit{LTL}_{\barwedge}(\neg\phi_2)$

Proof. $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \equiv \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\neg\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\neg\phi_2)$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \neg(\phi_1 \wedge \phi_2) \Leftrightarrow \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\neg\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\neg\phi_2)$. The translation rule $\mathrm{LTL}_{\barwedge}$ is only applied to $\neg(\phi_1 \wedge \phi_2)$ if $\neg(\phi_1 \wedge \phi_2)$ occurs in the formula $\phi_{1'} \barwedge_{<} \neg(\phi_1 \wedge \phi_2) \barwedge_{>} \phi_{2'}$ and, consequently,according to the semantics of EOL $\neg(\phi_1 \wedge \phi_2)$ has to hold on some $\sigma[l..l] \equiv s_l$.

$$s_l \vDash_e \neg(\phi_1 \wedge \phi_2) \quad \Leftrightarrow \quad s_l \vDash_l \mathrm{LTL}_{\barwedge}(\neg\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\neg\phi_2)$$

$$\begin{array}{c} s_l \vDash_e \neg(\phi_1 \wedge \phi_2) \textit{ iff} \\ s_l \vDash_e \neg\phi_1 \text{ and } s_l \vDash_e \neg\phi_2 \end{array} \Leftrightarrow \begin{array}{c} s_l \vDash_l \mathrm{LTL}_{\barwedge}(\neg\phi_1) \wedge \mathrm{LTL}_{\barwedge}(\neg\phi_2) \text{ iff} \\ s_l \vDash_l \mathrm{LTL}_{\barwedge}(\neg\phi_1) \text{ and } \mathrm{LTL}_{\barwedge}(\neg\phi_2) \end{array}$$

□

Theorem 37. $\sigma \vDash_e \psi_{\wedge_1} \wedge \psi_{\wedge 2} \equiv \sigma \vDash_l LTL_{\barwedge}(\psi_{\wedge_1}) \wedge LTL_{\barwedge}(\psi_{\wedge 2})$

Proof. $\sigma \vDash_e \psi_{\wedge_1} \wedge \psi_{\wedge 2} \equiv \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \wedge \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2})$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_{\wedge_1} \wedge \psi_{\wedge 2} \Leftrightarrow \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \wedge \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2})$

$$\sigma \vDash_e \psi_{\wedge_1} \wedge \psi_{\wedge 2} \quad \Leftrightarrow \quad \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \wedge \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2})$$

$$\begin{array}{c} \sigma \vDash_e \psi_{\wedge_1} \wedge \psi_{\wedge 2} \textit{ iff} \\ \sigma \vDash_e \psi_{\wedge_1} \text{ and } \sigma \vDash_e \psi_{\wedge 2} \end{array} \Leftrightarrow \begin{array}{c} \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \wedge \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2}) \text{ iff} \\ \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \text{ and } \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2}) \end{array}$$

□

Theorem 38. $\sigma \vDash_e \psi_{\wedge_1} \vee \psi_{\wedge 2} \equiv \sigma \vDash_l LTL_{\barwedge}(\psi_{\wedge_1}) \vee LTL_{\barwedge}(\psi_{\wedge 2})$

Proof. $\sigma \vDash_e \psi_{\wedge_1} \vee \psi_{\wedge 2} \equiv \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \vee \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2})$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \psi_{\wedge_1} \vee \psi_{\wedge 2} \Leftrightarrow \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \vee \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2})$

$$\sigma \vDash_e \psi_{\wedge_1} \vee \psi_{\wedge 2} \quad \Leftrightarrow \quad \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \vee \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2})$$

$$\begin{array}{c} \sigma \vDash_e \psi_{\wedge_1} \vee \psi_{\wedge 2} \textit{ iff} \\ \sigma \vDash_e \psi_{\wedge_1} \text{ or } \sigma \vDash_e \psi_{\wedge 2} \end{array} \Leftrightarrow \begin{array}{c} \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \vee \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2}) \text{ iff} \\ \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge_1}) \text{ or } \sigma \vDash_l \mathrm{LTL}_{\barwedge}(\psi_{\wedge 2}) \end{array}$$

□

Theorem 39. $\sigma \vDash_e \phi_1 \barwedge \phi_2 \equiv \Diamond(LTL_{\barwedge}(\phi_1) \wedge \Diamond LTL_{\barwedge}(\phi_2))$

Proof. $\sigma \vDash_e \phi_1 \barwedge \phi_2 \equiv \Diamond(\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \Diamond \mathrm{LTL}_{\barwedge}(\phi_2))$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi_1 \barwedge \phi_2 \Leftrightarrow \Diamond(\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \Diamond \mathrm{LTL}_{\barwedge}(\phi_2))$

$$\sigma \vDash_e \phi_1 \barwedge \phi_2 \quad \Leftrightarrow \quad \Diamond(\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \Diamond \mathrm{LTL}_{\barwedge}(\phi_2))$$

$$\begin{array}{c} \sigma \vDash_e \phi_1 \barwedge \phi_2 \textit{ iff} \\ \exists j,k : 0 \leq j < k \leq n \,.\, \sigma[0..j] \vDash_e \phi_1 \\ \text{and } \sigma[k..n] \vDash_e \phi_2 \end{array} \Leftrightarrow \begin{array}{c} \sigma \vDash_l \text{ true } \mathcal{U} \, (\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \\ \text{true } \mathcal{U} \, (\mathrm{LTL}_{\barwedge}(\phi_2))) \text{ iff} \\ \exists j \geq 0 \,. \\ \sigma[j...] \vDash_l (\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \\ \text{true } \mathcal{U} \, (\mathrm{LTL}_{\barwedge}(\phi_2))) \end{array}$$

$$\Leftrightarrow \begin{array}{c} \sigma \vDash_l \text{ true } \mathcal{U} \, (\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \\ \text{true } \mathcal{U} \, (\mathrm{LTL}_{\barwedge}(\phi_2))) \text{ iff} \\ \exists j \geq 0 \,.\, \sigma[j...] \vDash_l \mathrm{LTL}_{\barwedge}(\phi_1) \\ \text{and} \\ \exists k \geq j \,.\, \sigma[k...] \vDash_l \mathrm{LTL}_{\barwedge}(\phi_2) \end{array}$$

□

Theorem 40. $\sigma \vDash_e \phi_1 \barwedge_{[} \phi_2 \equiv \sigma \vDash_l \Diamond(LTL_{\barwedge}(\phi_1) \wedge \Box LTL_{\barwedge}(\phi_2))$

Proof. $\sigma \vDash_e \phi_1 \barwedge_{[} \phi_2 \equiv \sigma \vDash_l \Diamond(\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \Box \mathrm{LTL}_{\barwedge}(\phi_2))$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi_1 \barwedge_{[} \phi_2 \Leftrightarrow \sigma \vDash_l \Diamond(\mathrm{LTL}_{\barwedge}(\phi_1) \wedge \Box \mathrm{LTL}_{\barwedge}(\phi_2))$

$$\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_[} \phi_2 \quad \Leftrightarrow \quad \sigma \vDash_l \Diamond(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge \Box \mathrm{LTL}_{\hat{\wedge}}(\phi_2))$$

$$\begin{array}{c} \sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_[} \phi_2 \text{ iff} \\ \exists j : 0 \leq j \leq n \,.\, \sigma[j..j] \vDash_e \phi_1 \text{ and} \\ \forall k : j \leq k \leq n \,.\, \sigma[k..k] \vDash_e \phi_2 \end{array} \quad \Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \Diamond(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge \Box \mathrm{LTL}_{\hat{\wedge}}(\phi_2)) \text{ iff} \\ \text{true } \mathcal{U}(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge \neg(\text{ true } \mathcal{U} \neg \mathrm{LTL}_{\hat{\wedge}}(\phi_2))) \end{array}$$

$$\Leftrightarrow \quad \begin{array}{c} \exists j : j \geq 0 \,.\, \sigma[j..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi_1) \text{ and} \\ \forall k : k \geq j \,.\, \sigma[k..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi_2) \end{array}$$

□

Theorem 41. $\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_]} \phi_2 \equiv \mathit{LTL}_{\hat{\wedge}}(\phi_1)\, \mathcal{U}\, \mathit{LTL}_{\hat{\wedge}}(\phi_2)$

Proof. $\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_]} \phi_2 \equiv \mathrm{LTL}_{\hat{\wedge}}(\phi_1)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2)$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_]} \phi_2 \Leftrightarrow \mathrm{LTL}_{\hat{\wedge}}(\phi_1)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2)$

$$\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_]} \phi_2 \quad \Leftrightarrow \quad \mathrm{LTL}_{\hat{\wedge}}(\phi_1)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2)$$

$$\begin{array}{c} \sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_]} \phi_2 \text{ iff} \\ \exists j : i \leq j \leq r \,.\, \sigma[j..j] \vDash_e \psi \text{ and} \\ \forall k : 0 \leq k \leq j \,.\, \sigma[k..k] \vDash_e \phi_1 \end{array} \quad \Leftrightarrow \quad \begin{array}{c} \mathrm{LTL}_{\hat{\wedge}}(\phi_1)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2) \text{ iff} \\ \exists j : j \geq 0 \,.\, \sigma[j..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi_2) \text{ and} \\ \forall k : 0 \leq k \leq j \,.\, \sigma[k..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi_1) \end{array}$$

□

Theorem 42. $\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_<} \phi_\wedge \mathbin{\hat{\wedge}_>} \phi_2 \equiv \sigma \vDash_l \Diamond(\mathit{LTL}_{\hat{\wedge}}(\phi_1) \wedge (\mathit{LTL}_{\hat{\wedge}}(\phi_\wedge)\, \mathcal{U}\, \mathit{LTL}_{\hat{\wedge}}(\phi_2)))$

Proof. $\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_<} \phi_\wedge \mathbin{\hat{\wedge}_>} \phi_2 \equiv \sigma \vDash_l \Diamond(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge (\mathrm{LTL}_{\hat{\wedge}}(\phi_\wedge)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2)))$ holds if for any transition system T and all traces σ in T: $\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_<} \phi_\wedge \mathbin{\hat{\wedge}_>} \phi_2 \Leftrightarrow \sigma \vDash_l \Diamond(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge (\mathrm{LTL}_{\hat{\wedge}}(\phi_\wedge)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2)))$

$$\sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_<} \phi_\wedge \mathbin{\hat{\wedge}_>} \phi_2 \quad \Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \Diamond(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge \\ (\mathrm{LTL}_{\hat{\wedge}}(\phi_\wedge)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2))) \end{array}$$

$$\begin{array}{c} \sigma \vDash_e \phi_1 \mathbin{\hat{\wedge}_<} \phi_\wedge \mathbin{\hat{\wedge}_>} \phi_2 \text{ iff} \\ \exists j, k : i \leq j < k \leq r \,.\, \sigma[j..j] \vDash_e \phi_1 \\ \text{and } \sigma[k..r] \vDash_e \phi_2 \\ \text{and } \forall l : j \leq l \leq k \,.\, \sigma[l..l] \vDash_e \phi \end{array} \quad \Leftrightarrow \quad \begin{array}{c} \sigma \vDash_l \Diamond(\mathrm{LTL}_{\hat{\wedge}}(\phi_1) \wedge \\ (\mathrm{LTL}_{\hat{\wedge}}(\phi_\wedge)\, \mathcal{U}\, \mathrm{LTL}_{\hat{\wedge}}(\phi_2))) \text{ iff} \\ \exists j : j \leq 0 \,.\, \sigma[j..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi_1) \text{ and} \\ \exists k : k > j \,.\, \sigma[k..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi_2) \text{ and} \\ \forall l : j \leq l \leq k \,.\, \sigma[l..] \vDash_l \mathrm{LTL}_{\hat{\wedge}}(\phi) \end{array}$$

□

We have shown for all translation rules defined in Definition 27 that the LTL formula generated by the translation rule is equivalent, with respect to Definition 26, to the EOL formula from which it was generated.

Corollary 1. *It follows from Definition 27 and Theorems 29-42 that each EOL formula can be translated into an equivalent LTL formula.*

4.4 Relationship to ω-Automata

In Chapter 8 we leverage the fact that we can translate an EOL formula into an alternating ω-automata in order to limit the runtime intensive probability computation to the causal event combinations returned by the qualitative causality checking algorithm.

Automata are a common way to represent logic formulas. In this section we show how EOL formulas can be represented by alternating ω-automata.

It was shown in [99, 84] that each LTL formula can be translated into an alternating ω-automaton. From Corollary 1, which shows that each EOL formula can be translated into an equivalent LTL formula, it follows that we can translate each EOL formula into an alternating ω-automaton.

Given an EOL formula ψ in EONF we can construct an alternating automaton $A(\psi)$ such that $L(A(\psi)) = L(\psi)$. The construction of the automaton follows the structure of the formula.

Definition 28. *Alternating Automaton for an EOL formula. Let ψ an EOL formula that is built over the set of event variables $a \in \mathcal{A}$. The automaton $A(\psi)$ for the EOL formula ψ can be constructed recursively following the structure of the formula. Similarly to the LTL translation functions the translation function $A_{\owedge}(\psi)$ is used if ψ does contain one of the ordered operators $\owedge$, $\owedge_{[}$, $\owedge_{]}$, or $\owedge_{<}...\owedge_{>}$ and the translation function $A(\psi)$, else.*

$$
\begin{aligned}
A_{\owedge}(a) &= \langle a, \epsilon_A, + \rangle \\
A_{\owedge}(\neg a) &= \langle \neg a, \epsilon_A, + \rangle \\
A(a) &= \langle true, A(a), - \rangle \vee A_{\owedge}(a) \\
A(\neg a) &= \langle true, A(\neg a), + \rangle \wedge A_{\owedge}(\neg a) \\
A(\phi_1 \wedge \phi_2) &= A(\phi_1) \wedge A(\phi_2) \\
A(\neg(\phi_1 \wedge \phi_2)) &= A(\neg\phi_1) \wedge A(\neg\phi_2) \\
A_{\owedge}(\phi_1 \wedge \phi_2) &= A_{\owedge}(\phi_1) \wedge A_{\owedge}(\phi_2) \\
A_{\owedge}(\neg(\phi_1 \wedge \phi_2)) &= A_{\owedge}(\neg\phi_1) \wedge A_{\owedge}(\neg\phi_2) \\
A_{\owedge}(\psi_{\wedge_1} \wedge \psi_{\wedge_2}) &= A_{\owedge}(\psi_{\wedge_1}) \wedge A_{\owedge}(\psi_{\wedge_2}) \\
A_{\owedge}(\psi_{\wedge_1} \vee \psi_{\wedge_2}) &= A_{\owedge}(\psi_{\wedge_1}) \vee A_{\owedge}(\psi_{\wedge_2}) \\
A_{\owedge}(\phi_1 \owedge \phi_2) &= \langle true, A_{\owedge}(\phi_1 \owedge \phi_2), - \rangle \vee A_1 \quad \textit{where } A_1 = A_{\owedge}(\phi_1) \wedge A_2 \\
&\quad \textit{and } A_2 = \langle true, A_2, - \rangle \vee A_{\owedge}(\phi_2) \\
A(\phi_1 \owedge_{[} \phi_2) &= \langle true, A_{\owedge}(\phi_1 \owedge_{[} \phi_2), - \rangle \vee A_1 \quad \textit{where } A_1 = A_{\owedge}(\phi_1) \wedge A_2 \\
&\quad \textit{and } A_2 = \langle true, A_2, + \rangle \wedge A_{\owedge}(\phi_2) \\
A(\phi_1 \owedge_{]} \phi_2) &= A(\phi_2) \vee (\langle true, A(\phi_1 \owedge_{]} \phi_2), - \rangle \wedge A(\phi_1)) \\
A(\phi_1 \owedge_{<} \phi_{\wedge} \owedge_{>} \phi_2) &= \langle true, A(\phi_1 \owedge_{<} \phi_{\wedge} \owedge_{>} \phi_2), - \rangle \vee A_1 \ \textit{ where } A_1 = A_{\owedge}(\phi_1) \wedge A_2 \\
&\quad \textit{and } A_2 = A(\phi_2) \vee (\langle true, A_2), - \rangle \wedge A(\phi_{\wedge}))
\end{aligned}
$$

Corollary 2. *From Corollary 1 and Definition 17 (alternating automaton for an LTL formula) it immediately follows that for each EOL formula ψ there exists an alternating ω-automaton A for which $L(A(\psi)) = L(\psi)$ holds.*

Example 2. *To illustrate the proposed translation consider that for the EOL for-*

mula

$$\psi = (\, Ta \barwedge Gc)$$

of the railroad crossing example the first application of the recursive definition creates the following rewriting $A(\psi) = \langle true, A_{\barwedge}(\, Ta \barwedge Gc), - \rangle \vee (A_{\barwedge}(\, Ta) \wedge A_3)$ *and* $A_3 = \langle true, A_3, - \rangle \vee A_{\barwedge}(\, Gc)$*, Figure 4.2 shows the graphical representation of the automaton* $A(\psi)$*.*

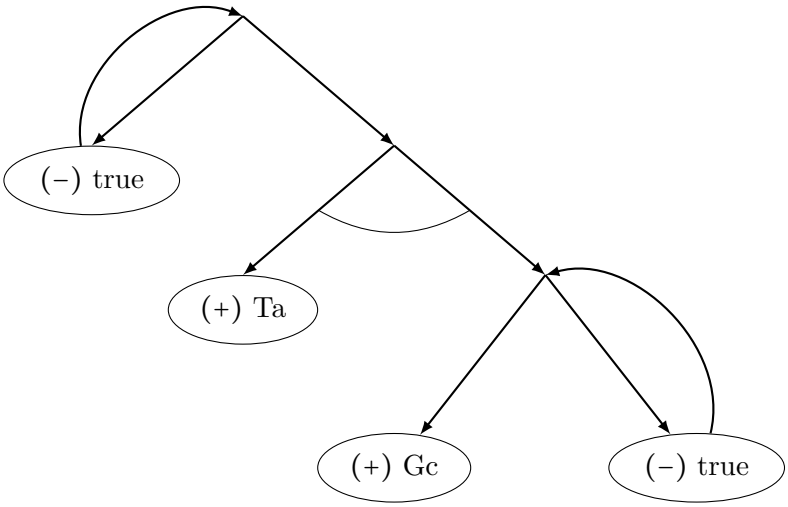


Figure 4.2: Graphical representation of the alternating automaton $A(\psi)$.

Chapter 5

Causality in System Models

The content of this chapter is based on the publications [61, 62, 76, 77, 78].

Contents

5.1 Introduction

In order to be able to reason about causal events, we need to first define the notion of causality that we want to base our reasoning on.

In the literature [57] two forms of causality are distinguished, the first one being *general type-level causality*, and the second one being *token-level causality*. General type-level causality deals with questions like, for example, "can random hardware failures cause a failure of the system?" while token-level causality provides answers to questions like, for instance, "is the failure of the gate in the railroad crossing causal for the crash between the car and the train?". In other words, general type-level causality uses statistical properties that may be learned from repeated observation of a system to derive causes and may be used to make predictions on the future occurrence of an effect after a cause has occurred. Token-level causality identifies the events that are causal for some particular effect and thus can be used to explain why some particular effect or hazard occurred. We want to provide information on which events caused a property violation. Consequently, we focus on token-level causality rather than focusing on general token-level causality.

A commonly used variant of token-level causality is the *counterfactual* reasoning argument and the related *alternative world* semantics of Lewis [32, 81]. The counterfactual argument is widely used as the foundation for identifying faults in program debugging [104] and also underlies the formal fault tree semantics proposed in [92]. 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.

The naive interpretation of the Lewis counterfactual test, however, leads to a number of inadequate or even fallacious inferences of causes as is discussed in [32]. The counterfactual argument in particular may lead to fallacious inferences if causes are given by logical conditions on the combinations of multiple events. We discuss the problematic issues, by demonstrating them with the railroad crossing example introduced in Section 3.2.

- Conjunction of Causal Events: In the railroad crossing both the car and the train have to be on the crossing to cause a crash. But since there are executions where only the train or the car are on the crossing and there is no crash the counterfactual argument fails to identify the conjunction of the two events as causes.

- Disjunction of Causal Events: Suppose that there are two scenarios for the gate in the railroad crossing to fail. The gate fails to close upon request or the gate closes with a delay when the car is already on the crossing. The counterfactual argument will not identify the gate failure as a cause for a crash because if the gate does not fail upon request there still can be a crash due to the delayed closing of the gate and vice versa.

- Preemption: If the event A happens but the effect B is preempted by, for instance, a repair mechanism the counterfactual test will not identify A as being a cause for B since there exists an execution where A occurs but B does not occur.

- Non-Occurrence of Events: The counterfactual test only reasons about the causality of the occurnce of an event but not about the non-occurence.

- Ordering of Events: Since we are considering concurrent systems in which particular event interleavings such as race conditions may be the cause of errors, the order of occurrence of events is a potential causal factor that cannot be disregarded. In the railroad crossing example we have the following event sequence "Cc, Gc, Tc, Tl, Go", where the car is on the crossing, the gate closes, the train is then on the crossing and leaving the crossing and the gate is opening. This sequence clearly leads to a crash since the car and the train are on the crossing at the same time. The second event sequence "Gc, Tc, Tl, Go, Cc", consisting of the same events, in a different order does not lead to a crash because the train leaves the crossing before the car enters the crossing. For the counterfactual argument to be applicable the constraint that whenever the causal events occur the hazard occurs as well has to hold. But on the above traces the same events occur, but the hazard occurs only on one trace, consequently, the counterfactual argument fails to identify the first sequence as a causal sequence of events.

- Irrelevance of Events: In addition, the naive counterfactual test may determine irrelevant causal events. For instance, the fact that the union of train

> engineers has decided not to call for a strike is not to be considered a cause for the occurrence of an accident at the railroad crossing, but since if the union would have called for a strike the accident would not have happened the counterfactual argument would identify this decision as causal.

Halpern and Pearl extend the Lewis counterfactual model to what they refer to as *structural equation model* (SEM) [46]. It encompasses the notion of *actual causes* being logical combinations of events as well as a distinction of relevant and irrelevant causes. In the SEM events are represented by variable values and the minimal number of causal variable valuation combinations is determined. In order to do so the counterfactual test is extended by contingencies. Contingencies can be viewed as possible alternative worlds, where a variable value is changed. A variable X is causal if there exists a contingency, that is a variable valuation for other variables, that makes X counterfactual. However, the structural equation model does not account for event orderings, which is a major concern of this thesis.

We now sketch the structural equation and actual cause definition from [46]. Structural equations are used to described the causal influence of variables representing the occurrence of events on other variables representing the occurrence of events. The set of all variables is partitioned into the set U of *exogenous* variables and the set V of *endogenous* variables. Exogenous variables represent facts that we do not consider to be causal factors for the effect that we analyze, even though we need to have a formal representation for them so as to encode the "context" ([46]) in which we perform causal analysis. An example for an exogenous variable is the decision of the union of train engineers in the above railroad crossing example. Endogenous variables represent all events that are considered to have a meaningful, potentially causal effect. The set $X \subseteq V$ contains all events that are expected jointly to be a candidate cause. More formally Halpern and Pearl define a signature $\mathcal{S}$ as a tuple $(\mathcal{U}, \mathcal{V}, \mathcal{R})$, where $\mathcal{U}$ is a finite set of exogenous variables, $\mathcal{V}$ is a finite set of endogenous variables, and $\mathcal{R}$ associates with every variable $Y \in \mathcal{U} \cup \mathcal{V}$ a nonempty set $\mathcal{R}(Y)$ of possible values for Y. A structural equation model over a signature $\mathcal{S}$ is defined in [46] as tuple $M = (\mathcal{S}, \mathcal{F})$, where $\mathcal{F}$ associates with each variable $X \in \mathcal{V}$ a function denoted F_X that defines the values of all variables in X given the values of all other variables in $\mathcal{U} \cup \mathcal{V}$. Given a structural equation model $M = (\mathcal{S}, \mathcal{F})$, a (possibly empty) vector $\vec{X}$ of variables in $\mathcal{V}$, and vectors $\vec{x}$ and $\vec{u}$ of values for the variables in $\vec{X}$ and $\mathcal{U}$, a new structural model denoted by $M_{\vec{X} \leftarrow \vec{x}}$ over the signature $\mathcal{S}_{\vec{X}} = (\mathcal{U}, \mathcal{V} - \vec{X}, \mathcal{R}|_{\mathcal{V} - \vec{X}})$ is defined. $M_{\vec{X} \leftarrow \vec{x}}$ is the structural equation model that results when the variables in $\vec{X}$ are set to $\vec{x}$. Halpern and Pearl further define that, given a signature $\mathcal{S} = (\mathcal{U}, \mathcal{V}, \mathcal{R})$, a formula of the form $X = x$, for $X \in \mathcal{V}$ and $x \in \mathcal{R}(X)$, is called a primitive event. A basic causal formula over $\mathcal{S}$ is one of the form $[Y_1 \leftarrow y_1, ..., Y_k \leftarrow y_k,]\varphi$ where φ is characterizing the hazard of effect and $Y_1, ...Y_k$ and X are variables in $\mathcal{V}$. The formula $[Y_1 \leftarrow y_1, ..., Y_k \leftarrow y_k,]\varphi$ is abbreviated as $[\vec{Y} \leftarrow \vec{y}]\varphi$. Intuitively, $[\vec{Y} \leftarrow \vec{y}]\varphi$ states that φ holds in an alternative world that is defined by setting the values of the variables in $\vec{Y}$ to the values defined in $\vec{y}$. A causal formula ψ is a Boolean combination of basic causal formulas and ψ is

true or false in a structural equation model, given the context defined by values of the variables in $\mathcal{U}$. If ψ is true in a structural model given the context defined by $\vec{u}$ this is is written as $(M, \vec{u}) \models_{SM} \psi$.

The types of events that Halpern and Pearl allow as actual causes are conjunctions of primitive events of the form $X_1 = x_1 \wedge .... \wedge X_k = x_k$ that are abbreviated as $\vec{X} = \vec{x}$. Formally Halpern and Pearl define an actual cause as follows:

Definition 29. *Actual cause defined by Halpern and Pearl in [46]* $\vec{X} = \vec{x}$ *is an actual cause of* φ *in* $(M, \vec{u})$ *if the following three actual cause conditions (AC) hold:*

AC1: $(M, \vec{u}) \models_{SM} (\vec{X} = \vec{x}) \wedge \varphi$. *In words, both* $\vec{X} = \vec{x}$ *and* φ *are true in the actual world, which means that the variables in* $\vec{X}$ *have the values specified by* $\vec{x}$ *and the effect or hazard specified by* φ *occurs.*

AC2: *There exists a partition* $(\vec{Z}, \vec{W})$ *of* $\mathcal{V}$ *with* $\vec{X} \subseteq \vec{Z}$ *and some setting* $(\vec{x}, \vec{w})$ *of the variable in* $(\vec{X}, \vec{W})$ *such that:*

1. $(M, \vec{u}) \models_{SM} [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}']\neg\varphi$, *which means that changing the values of the variables in* $(\vec{X}, \vec{W})$ *from the values specified by* $(\vec{x}, \vec{w})$ *to some other values specified by* $(\vec{x}', \vec{w}')$ *changes* φ *form true to false.*
2. $(M, \vec{u}) \models_{SM} [\vec{X} \leftarrow \vec{x}, \vec{W} \leftarrow \vec{w}', \vec{Z} \leftarrow \vec{z}^*]\varphi$ *for all subsets* $\vec{Z}'$ *of* $\vec{Z}$, *which means that changing the values of the variables* $\vec{W}$ *from the values specified by* $\vec{w}$ *to some other values specified by* $\vec{w}'$ *should have no effect on* φ *as long as the values of the variables in* $\vec{X}$ *are kept at the values specified by* $\vec{x}$, *even if all the variable values in an arbitrary subset of* $\vec{Z}$ *are set to their initial values.*

AC3: $\vec{X}$ *is minimal, with respect to that no subset of* $\vec{X}$ *satisfies conditions AC1 and AC2. Minimality ensures that only those elements of the conjunction* $\vec{X} = \vec{x}$ *that are essential for changing* φ *in AC2(1) are considered as part of the cause and that all inessential elements are pruned.*

In summary an actual cause according to Halpern and Pearl is a causal formula from which irrelevant events have been removed. A causal formula is a boolean conjunction ψ of variables representing the occurrence of events.

The condition AC2(1) corresponds to the Lewis counterfactual test. However, as [46] argues, AC2(1) is too permissive in the sense that it allows to change the values of the variables in X and in W and, thus, it is not clear whether the change of φ form true to false was caused by change a variable in X or by changing a variable in W. AC2(2) constrains what is admitted as cause by AC2(1) by keeping the values of the variables in X at their original values and only changing the variables in W. Minimality in AC3 ensures that only those elements of the conjunction that are essential for changing φ in AC2(1) are considered part of the cause and that all inessential elements are pruned from the causal formula.

5.2 Inferring Causality in System Models

The structural equation model approach and the actual cause definition by Halpern and Pearl that we have introduced in the previous section, can not be directly applied to system models. The main reason for this is that in the actual cause definition arbitrary changes can be made to the variable values representing the event occurrences whereas in system models the occurrence of events is limited by the executions that are possible in the system model. Another reason why we can not directly apply the Halpern and Pearl structural equation model and the actual cause definition is that it does not consider the order of the occurrences of events as possible causal factors. In order for the actual cause definition to be applicable to system models a number of adaptations need to be made.

We do not account for exogenous variables, since we assume that everything that is modeled in the system model is relevant and thus endogenous variables. However, should one wish to consider exogenous variables, those can easily be retrofitted.

We only consider boolean variables, and the variable associated with an event is true in case that event has occurred. Notice that the use of this operational form of event semantics makes the use of structural equation model to define events as in [46] dispensable. In other words, we inherit from Halpern and Pearl the general ideas of the actual cause definitions, but not the structural equation model based event semantics.

The boolean conjunction of the event variables forms a causal event order logic formula ψ. In the actual cause definition by Halpern and Pearl it is possible to change the values of the variables in X and Z independently from each other. The variables in the set Z, of which X is a subset, describe what Halpern and Pearl call a causal process. The causal process comprises all variables that mediate between the events represented by the variables in X and the effect φ. Not all variables in Z are root-causes, but they contribute to rippling the causal effect through the system until reaching the final effect. Since when analyzing system models we are limited to the behavior specified by the model it is not possible to change the values of the variables in X and Z independently from each other, because there always has to exist a corresponding execution trace. For this reason we limit our actual cause definition to detect the events in the set Z, that includes the variables in X, and comprises the complete causal process for some effect.

We adapt the actual cause definition by Halpern and Pearl, introduced in Section 5.1, such that it can be used to decide whether a given EOL formula ψ describes the causal process of the violation of some LTL formula φ in a transition system T. Note that in this thesis we restrict the causality reasoning to the violation of non-reachability properties. Hence we only need to consider finite execution traces [8]. Furthermore, we extend the actual cause definition to consider the order of the occurrences of events as possible causal factors. The causal process comprises the causal events for the property violation and all events that mediate between the causal events and the property violation. Those events which are not root-causes, are needed to propagate the cause through the system until the property violation

is being triggered. If ψ describes the causal process of a property violation we also say ψ is causal for the property violation.

In a naive causality checking algorithm we perform the tests defined in Definition 30 for the induced EOL formula ψ_σ of each $\sigma \in \Sigma_B$. The disjunction of all $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_n}$ that satisfy the conditions AC1-AC3 form the EOL formula Ψ describing all possible causes of the hazard.

Definition 30. *Cause for a Property Violation (adapted actual cause). Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and σ, σ' and σ'' some execution traces of T. An EOL formula ψ containing all the event variables in Z is considered a cause for an effect represented by the violation of the LTL non-reachability property φ, if the following conditions are satisfied: We partition the set of event variables $\mathcal{A}$ into sets Z and W, such that Z contains all event variables that are part of the EOL formula ψ and let $W = \mathcal{A} \setminus Z$. We use the function $val_\mathcal{A}(\sigma)$, defined in Section 4.2 in Definition 19, to represent the valuation of all variables in $\mathcal{A}$ for a given σ.*

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

Example 3. *If we want, for instance, to show that $\psi = Ta \wedge Ca \wedge Gf \wedge Cc \wedge Tc$ is causal, we need to show that AC1, AC2(1), AC2(2) and AC3 are fulfilled for ψ.*

- *AC1 is fulfilled, since there exists an execution σ = "Ta, Ca, Gf, Cc, Tc" for which $\sigma \vDash_e \psi$ holds, and both the train and the car are in the crossing at the same time.*
- *AC2(1) is fulfilled since there exists an execution σ' = "Ta, Ca, Gc, Tc" for which $\sigma' \nvDash_e \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \wedge val_W(\sigma) \neq val_W(\sigma'))$ holds and σ' does not violate the property. In this case the sets Z and W are $Z = \{Ta, Ca, Gf, Tc\}$ and $W = \{Gc, Tl, Cl, Go\}$. The returned valuations by the valuation function for Z, W and σ, σ' are $val_Z(\sigma) = (true, true, true, true)$, $val_Z(\sigma') = (true, true, false, true)$, $val_W(\sigma) = (false, false, false, false)$, and $val_W(\sigma') = (true, false, false, false)$.*
- *Now we need to check the condition AC2(2). For the execution σ'' ="Ta, Ca, Gf, Cc, Cl, Tc" and the partition $Z, W \subseteq \mathcal{A}$, $\sigma'' \vDash_e \psi$ and $val_Z(\sigma) =$*

$val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma'')$ hold. In this case the sets Z and W are $Z = \{Ta, Ca, Gf, Tc\}$ and $W = \{Gc, Tl, Cl, Go\}$. The returned valuations by the valuation function for Z, W and σ, σ' are $val_Z(\sigma) = (true, true, true, true)$, $val_Z(\sigma'') = (true, true, true, true)$, $val_W(\sigma) = (false, false, false, false)$, and $val_W(\sigma') = (false, false, true, false)$. The property is not violated since the car leaves the crossing (Cl) before the train enters the crossing (Tc). As a consequence, AC2(2) is not fulfilled by ψ because if Cl occurs between Cc and Tc, the property violation is prevented.

Example 3 shows that the non-occurrence of events can be causal as well, and that this is not yet captured by the adapted actual cause definition. The non-occurrence of an event is causal whenever AC1 and AC2(1) are fulfilled but AC2(2) fails for a EOL formula ψ_σ. If AC2(2) fails there is at least one event α on σ'' which did not occur on σ and the occurrence of α prevents the property violation. Consequently, the non-occurrence of α on σ is causal. We need to reflect the causal effect of the non-occurrence of α in ψ_σ. For the models that we analyze there are two possibilities for such a preventing event α to occur, namely,

1. at the beginning of the execution trace, or
2. between two other events α_1 and α_2.

Note that since we are only analyzing the causal events for the violation of non-reachability properties an event preventing a property violation can never occur at the end of the execution trace. Once the property is violated, it can not be prevented by any occurrence of a future event.

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

Definition 31. *Non-Occurrence of Events. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and σ and σ'' execution traces of T. We partition the set of event variables $\mathcal{A}$ into sets Z and W, such that Z contains all event variables that are part of the EOL formula ψ and let $W = \mathcal{A} \setminus Z$. The non-occurrence of the events which are represented by the event variables $a_\alpha \in Q$ with $Q \subseteq W$ on execution σ is causal for the violation of the LTL formula φ if ψ satisfies AC1 and AC2(1) but violates AC2(2), and if Q is minimal, which means that there is no true subset of Q for which $\sigma'' \vDash_e \psi \wedge val_Z(\sigma) = val_Z(\sigma'') \wedge val_Q(\sigma) \neq val_Q(\sigma'') \wedge val_{W \setminus Q}(\sigma) = val_{W \setminus Q}(\sigma'')$ and $\sigma'' \nvDash_l \varphi$ holds. In addition we require that for no other set Q' that satisfies conditions described above $Q' \subseteq Q$.*

For each Q we determine the location of the event variables $a_\alpha \in Q$ in ψ'' and prohibit the occurrence of α in the same location in ψ. If there are more than one event variables in Q, they are connected with an $\wedge$-operator. If there

are more than one set of events at one location in ψ they are connected by an $\vee$-operator. Thus we obtain a simple EOL formula ϕ specifying the events of which their non-occurrence is causal. We add $\neg\phi\barwedge_]$ at the beginning of ψ if the events occurred at the beginning of σ'' and if the events occurred between the two events α_1 and α_2 we insert $\barwedge_<\neg\phi\barwedge_>$ between the two event variables a_{α_1} and a_{α_2} in ψ. Additionally, each event variable in Q is added to Z. Definition 31 identifies the set Q of event variables that conjunctively prevent the occurrence of the property violation. It is possible that more than one minimal set of event variables that prevents the property violation is found by Definition 31, if this is the case it is checked whether both sets are minimal and if that is the case they can disjunctively prevent the property violation. Consequently, Definition 31 will identify the minimal conjunctions and disjunctions of events that prevent a property violation. In our example, Cl is the only event that can prevent the property violation on σ and occurs between the events Cc and Tc. Consequently, $\neg Cl$ is added to Z and ψ and we get ψ = Ta $\barwedge$ Ca $\barwedge$ Gf $\barwedge$ Cc $\barwedge_<$ $\neg$Cl $\barwedge_>$ Tc.

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

Definition 32. *Order Condition (OC). Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and σ, σ' execution traces of T. Let ψ an EOL formula over Z that holds for σ and let $\psi_\wedge$ the EOL formula that is created by replacing all $\barwedge$-operators in ψ by $\wedge$-operators. The $\barwedge_[$, $\barwedge_]$, and $\barwedge_< \phi \barwedge_>$ are not replaced in $\psi_\wedge$.*

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

If the order of a subset of events in ψ_σ is not cause, the condition OC relaxes the order constraint by replacing the $\barwedge$-operator with the $\wedge$-operator in such a way that all possible interleavings of σ that cause a violation of the property are represented by ψ_σ.

In our example, the order of the events Gf, Cc, $\neg$Cl, Tc is causal since an accident only happens 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. Consequently, after OC we obtain the EOL formula ψ = Gf $\wedge$ ((Ta $\wedge$ (Ca $\barwedge$ Cc)) $\barwedge_<$ $\neg$Cl $\barwedge_>$ Tc).

The disjunction of all $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_n}$ that satisfy the conditions AC1-AC3 and OC is the EOL formula $\Psi = \psi_{\sigma_1} \vee \psi_{\sigma_2} \vee ... \vee \psi_{\sigma_n}$ describing all possible causes of the hazard. It is possible that there are duplicate ψ_σ generated by the OC test from

different interleavings of σ. In this case only one ψ_σ is added as a disjunct to Ψ. For the railroad crossing example this EOL formula is $\Psi = (\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \curlywedge \mathrm{Cc})) \curlywedge_< \neg\mathrm{Cl} \curlywedge_> \mathrm{Tc})) \vee ((\mathrm{Ta} \wedge (\mathrm{Ca} \curlywedge \mathrm{Cc})) \curlywedge_< \neg\mathrm{Cl} \curlywedge_> (\mathrm{Gc} \wedge \mathrm{Tc}))$. Intuitively, each disjunct of this formula represents a class of execution traces on which the events specified by the EOL formula cause the violation of the property. The two classes represented by the disjuncts are:

1. If the gate fails (Gf) at some point of the execution and both a train (Ta) and a car (Ca) are approaching this results in a hazardous situation if the car is on the crossing (Cc) and does not leave the crossing (Cl) before the train (Tc) enters the crossing $(\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \curlywedge \mathrm{Cc})) \curlywedge_< \neg\mathrm{Cl} \curlywedge_> \mathrm{Tc}))$.

2. If both a train (Ta) and a car (Ca) are approaching but the gate closes (Gc) when the car (Cc) is already on the railroad crossing and is not able to leave (Cl) before the gate is closing and the train is crossing (Tc), this also corresponds to a hazardous situation $((\mathrm{Ta} \wedge (\mathrm{Ca} \curlywedge \mathrm{Cc})) \curlywedge_< \neg\mathrm{Cl} \curlywedge_> (\mathrm{Gc} \wedge \mathrm{Tc}))$.

For instance, the execution traces $\sigma = \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Tc}$ and $\sigma' = \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gc}, \mathrm{Tc}, \mathrm{Tl}, \mathrm{Go}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Tc}$ are traces that belong to the first class of traces. The trace $\sigma'' = \mathrm{Ca}, \mathrm{Ta}, \mathrm{Cc}, \mathrm{Gc}, \mathrm{Tc}$ is an example for a trace in the second class.

We now formalize the observation that each disjunct of an EOL formula represents a class of traces by introducing the notion of causality classes.

Definition 33. *Causality Class. Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system and $\sigma = s_0, \alpha_1, s_1, \alpha_2, \ldots \alpha_n, s_n$ a finite execution trace of T. The set Σ_B is the set of traces for which some LTL non-reachability property φ is violated.*

The causality classes $CC_1, ..., CC_n$ defined by the disjuncts of the EOL formula $\Psi = \psi_1 \vee ... \vee \psi_n$, satisfying Definition 30, Definition 31, and Definition 32, decompose the set Σ_B into sets $\Sigma_{B_{\psi_1}}, ..., \Sigma_{B_{\psi_n}}$ with $\Sigma_{B_{\psi_1}} \cup ... \cup \Sigma_{B_{\psi_n}} = \Sigma_B$.

Note that it can be the case that $\sigma \in \Sigma_{B_{\psi_1}} \wedge \sigma \in \Sigma_{B_{\psi_2}}$ if $\sigma \vDash_e \psi_1 \wedge \sigma \vDash_e \psi_2$, which has to be taken into account if the probabilities of the causality classes are computed.

5.3 Completeness and Soundness

In this section we discuss the completeness and soundness of the proposed causality checking approach.

5.3.1 Completeness

Before we are able to discuss whether causality checking is complete or not, we need to define what the term *completeness* means with respect to a causality checking result. We base this discussion on the notion of causality classes defined in Definition 33.

There are two interpretations of completeness in this setting:

1. For each possible combination of events, that is possible orderings and occurrences of events, that can cause a property violation in the system there exists a causality class or a combination of causality classes capturing this combination of events.

2. For each possible bad trace in the system model there also exists a causality class representing this trace.

The first notion of completeness requires that all possible event combinations that are possible in the real system, are also possible in the system model. This requires that all possible event combinations that can cause a property violation are modeled in the system model. We assume that the system models we analyze are complete.

If the system model is complete, there exists an execution trace for each possible combination of events that can cause a property violation. Consequently, the completeness of causality checking depends on a complete enumeration of all bad and good traces in the system model by the model checking algorithms used for causality checking. We discuss here the completeness of the naive causality checking approach where we assume that all bad and good traces are identified, for the approaches in Chapters 6-8 we will later show that this assumption can be guaranteed.

Since we assume that all bad and good traces in the system model are enumerated, we can assume that for each possible combination of events that can cause a property violation there exists a bad trace consisting of those events. Consequently, it suffices to show that causality checking is complete so that for each possible bad trace in the system there also exists a causality class representing this trace. Therefore, we have to show that each possible bad trace in the system is represented by one of the disjuncts of the event order logic formula Ψ returned by the causality checker.

Theorem 43. *If all bad traces have been identified, the event order logic formula returned by the causality checker is complete, so that there does not exist a trace leading to a property violation that is not captured by the event order logic formula Ψ which is returned by the causality checker. Formally we define $\forall \sigma \in T \,.\, \sigma \not\models_l \varphi \Rightarrow \sigma \models_e \Psi$.*

Proof. We assume that there exists a bad trace σ for which $\sigma \not\models_e \Psi$ holds. For each bad trace σ it is checked whether the EOL formula ψ_σ representing this bad trace shall be added as a disjunct to the formula Ψ. A EOL formula ψ_σ is added to Ψ if it satisfies the tests AC1 through AC3. Therefore, for a bad trace σ not to be captured by one of the disjuncts of Ψ, ψ_σ needs to be excluded from Ψ by one of the AC1-AC3 tests or the OC test.

- The test AC1 excludes an EOL formula ψ_σ representing a trace σ if no property violation occurs on σ, which contradicts our assumption that σ is a bad trace.

- For ψ_σ to be excluded by the test AC2(1) it is necessary that no execution trace $\sigma' \in \Sigma_G$ where the order and occurrence of the events is different from σ exists. For each bad trace σ there exists a finite prefix $\sigma' \in \Sigma_G$ of σ and the order and occurrence of the events on σ' is different from σ, therefore, AC2(1) is satisfied for all bad traces and, consequently, no ψ_σ representing a bad trace σ is excluded from from Ψ by the AC2(1) test.

- If the test AC2(2) fails for a ψ_σ representing a bad trace σ, σ is not excluded but ψ_σ is altered according to Definition 31. The constraints introduced to ψ_σ by Definition 31 only constrain events not occurring on σ. Consequently, $\sigma \vDash_e \psi_\sigma$ still holds for the altered ψ_σ and thus no bad trace is excluded from Ψ by AC2(2).

- For the test AC3 to exclude ψ_σ representing a bad trace σ from Ψ, there has to exist a trace $\sigma'' \in \Sigma_B$ that implies a violation of the minimality condition for ψ_σ. Since the EOL formula $\psi_{\sigma''}$ representing σ'' is then part of Ψ and it holds that $\sigma \vDash_e \psi_{\sigma''}$, σ is still represented by a disjunct of Ψ.

- The OC test does not exclude any ψ_σ from Ψ, but merely relaxes the order constraints of a ψ_σ, by replacing the $\barwedge$-operator with the $\wedge$-operator, in order to represent all interleavings of σ that are also bad traces. If ψ_σ is part of Ψ the trace σ can not be excluded by OC because even if all $\barwedge$-operators where replaced by $\wedge$-operators $\psi_\sigma \vDash_e \sigma$ holds.

As we have shown above it is not possible that there exists a bad trace which is not represented by a disjunct of Ψ. Consequently, the EOL formula Ψ returned by the causality checker is complete. □

5.3.2 Soundness

We define a causality checking result to be sound if whenever the events described by a causality classes occur, the property violation occurs.

We assume that the bad traces we take as an input for the naive causality checking approach are sound. We will later show how this assumption can be guaranteed for the approaches in Chapters 6-8.

We need to show that on all traces satisfying the EOL formula Ψ, which is returned by the causality checker, the property is violated.

Theorem 44. *If all good traces have been found, the event order logic formula Ψ returned by the causality checker is sound, so that on each trace σ for which $\sigma \vDash_e \Psi$ holds indeed the property is violated.*

Proof. We assume that there exists a trace σ satisfying a causality class ψ of Ψ and on σ the property φ is not violated.

There are two cases that we need to consider:

1. ψ is added to Ψ because AC1-AC3 are satisfied for ψ. If $\sigma \vDash \psi$ is true, the event combination specified by ψ occurs on σ. For σ being a good trace, this would require that the property violation is prevented by some event on σ which is not constraint by ψ. If such an event exists, the AC2(2) test fails and ψ would not have been added to Ψ and instead an altered version of ψ ensuring the non-occurrence of the event preventing the property violation.

2. If ψ was added to Ψ because of the bad trace σ' and σ' is an interleaving of σ, the order constraints of ψ are changed by the OC test to also represent σ. But the OC test will not change the EOL formula ψ representing σ' in such a way that the order of events on a good trace σ are accepted, because only the event orderings of bad traces are considered for the OC test.

Therefore, we have shown that it can not be the case that there exists a trace $\sigma \vDash_e \Psi$ on which the property is not violated and have thus proven the soundness of the causality checking result. □

Chapter 6

Qualitative Causality Checking

The content of this chapter is based on the publications [13, 14, 74, 75, 76, 80].

Contents

6.1 Introduction

We have established the formal basis to reason about event occurrences and their order and have defined a notion of causality in system models. This chapter discusses the integration of causality reasoning into the state-space exploration algorithms used for model checking. The *causality checking* approach proposed in Section 6.2 can be integrated with both depth-first search (DFS) and breadth-first search (BFS) as shown in Section 6.3. In Section 6.4 we discuss the completeness and soundness of the proposed causality checking approach. Complexity considerations of the causality checking approach are discussed in Section 6.5. We evaluate our approach with respect to runtime and memory consumption as well as usefulness of the results on several case-studies in Section 6.6.

6.2 Causality Checking

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

The key idea of the proposed algorithm is that the conditions AC1, AC2(1), AC2(2) and AC3 defined in Chapter 5 can be mapped to computing sub- and superset relationships between good and bad execution traces. We define a number of execution trace comparison operators as follows.

Definition 34. *Execution Trace Comparison Operators. Let* $T = (S, Act, \rightarrow, I, AP, L)$ *a transition system, and* σ_1 *and* σ_2 *execution traces of T.*

$=$**:** $\sigma_1 = \sigma_2$ *iff* $\forall a \in \mathcal{A} \,.\, \sigma_1 \vDash_e a \equiv \sigma_2 \vDash_e a$.

$\doteq$**:** $\sigma_1 \doteq \sigma_2$ *iff* $\sigma_1 = \sigma_2$ *and* $\forall a_1, a_2 \in \mathcal{A} \,.\, \sigma_1 \vDash_e a_1 \wedge a_2 \equiv \sigma_2 \vDash_e a_1 \wedge a_2$.

$\subseteq$**:** $\sigma_1 \subseteq \sigma_2$ *iff* $\forall a \in \mathcal{A} \,.\, \sigma_1 \vDash_e a \Rightarrow \sigma_2 \vDash_e a$.

$\subset$**:** $\sigma_1 \subset \sigma_2$ *iff* $\sigma_1 \subseteq \sigma_2$ *and not* $\sigma_1 = \sigma_2$.

$\dot{\subseteq}$**:** $\sigma_1 \dot{\subseteq} \sigma_2$ *iff* $\sigma_1 \subseteq \sigma_2$ *and* $\forall a_1, a_2 \in \mathcal{A} \,.\, \sigma_1 \vDash_e a_1 \wedge a_2 \Rightarrow \sigma_2 \vDash_e a_1 \wedge a_2$.

$\dot{\subset}$**:** $\sigma_1 \dot{\subset} \sigma_2$ *iff* $\sigma_1 \dot{\subseteq} \sigma_2$ *and not* $\sigma_1 \doteq \sigma_2$.

For the empty trace $\sigma_0 = \{\}$*:* $\sigma_0 = \sigma_1$ *iff* $\sigma_1 = \{\}$, $\sigma_0 \doteq \sigma_1$ *iff* $\sigma_1 = \{\}$, $\sigma_0 \subseteq \sigma_1$ *is true for all* σ_1, $\sigma_0 \subset \sigma_1$ *is true for all* $\sigma_1 \neq \{\}$, $\sigma_0 \dot{\subseteq} \sigma_1$ *is true for all* σ_1, $\sigma_0 \dot{\subset} \sigma_1$ *is true for all* $\sigma_1 \neq \{\}$.

For the traces σ = "Ta, Ca" and σ' = "Ca, Ta", for instance, $\sigma = \sigma'$, $\sigma \subseteq \sigma'$ and $\sigma \neq \sigma'$ hold. In the following we also use the terms sub-execution and super-execution to refer to sub- or superset relationships between execution traces.

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

Theorem 45. *AC1 is fulfilled for all* ψ_σ *where* $\sigma \in \Sigma_B$.

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

Theorem 46. *AC2(1) holds for* ψ_σ *if there is an execution* $\sigma' \in \Sigma_G$ *with* $\sigma' \subset \sigma$.

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

Theorem 47. *AC2(2) holds for* ψ_σ *if there is no execution* $\sigma'' \in \Sigma_G$ *with* $\sigma \dot{\subset} \sigma''$.

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

Theorem 48. *If AC1 and AC2(1) hold for ψ_σ and ψ_σ is modified according to Definition 31 in order to fulfill AC2(2), then AC1 and AC2(1) hold for the modified ψ_σ.*

Proof. The modification defined in Definition 31 prohibits the occurrence of events that did not occur on σ but occur on σ'' by adding their corresponding negated event variables to ψ_σ. Since the prohibited events did not occur on σ, the modified ψ_σ holds for σ and AC1 holds. AC2(1) holds for the modified ψ_σ because for AC2(1) to hold in the first place there has to be an execution $\sigma' \in \Sigma_G$ with $\sigma' \subset \sigma$. For the modification of ψ_σ to be necessary an execution $\sigma'' \in \Sigma_G$ with $\sigma \dot{\subset} \sigma''$ has to exist. If $\sigma \dot{\subset} \sigma''$ holds, $\sigma \subset \sigma''$ holds and $\sigma' \subset \sigma''$ holds as well. Consequently, AC2(1) holds for the modified ψ_σ. □

Theorem 49. *AC(3) holds for ψ_σ if there does not exists an execution $\sigma''' \in \Sigma_B$ for which $\sigma''' \subset \sigma$ holds.*

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

We use these theorems in order to devise an algorithm and a corresponding data structure called subset graph for on-the-fly causality checking.

6.3 Integration into State-Space Exploration

The causality checking that we propose is embedded into both of the standard state-space exploration algorithms used in explicit state model checking, namely

depth-first search (DFS) and breadth-first search (BFS). Whenever a bad or a good execution is found by the search algorithm it is stored in the prefix-tree data structure described in Section 6.3.1 and the corresponding sub- and superset relationships with other traces are stored in the subset graph data structure described in Section 6.3.2.

6.3.1 Prefix Tree

In order to efficiently store the execution traces, we use two prefix tree [43] data structures. The first prefix tree data structure stores the actions representing the events of the execution traces and the second prefix tree data structure stores the states of the execution traces. The prefix tree is an efficient method for the storage of the execution traces, since for each trace with a length greater than one we already have stored its prefix. For the traces "Ta", "Ta, Ca", and "Ta, Ca, Gc", for instance, we only need to store "Ta", "Ca", and "Gc" and the corresponding prefix relationship instead of storing all three execution traces, individually.

6.3.2 Subset Graph Data Structure

In order to store the sub- and superset relationships of the execution traces we have devised a data structure called subset graph. This data structure enables us to make causality decisions on-the-fly which means that we can decide whether an execution trace is causal as soon as we add it to the subset graph.

The subset graph is structured into levels where each level corresponds to the length of the execution traces stored on that level. Each node represents exactly one execution trace. Figure 6.1 shows a part of the subset graph for the railroad crossing example. The execution traces on adjoining levels are connected by edges indicating subset relationships between the respective execution traces. To improve readability the edges between executions on the same level are not displayed in the figure.

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

- Green: a node is colored *green* if it represents a good execution trace and all nodes on the level below that are connected with it are also colored green. An example of such a trace is "Ca,Ta,Gc,Tc,Tl" in the railroad crossing example. Green traces can not be causal because they are good traces. The green traces can be prefixes of either bad or good execution traces.

- Red: a node is colored *red* if it represents a bad execution trace and all nodes on the level below that are connected with it are colored green. Red nodes correspond to the shortest bad traces found at any point of the state-space exploration. They are considered to be causal. As an example consider the trace "Ta,Ca,Gf,Cc,Tc" in the railroad crossing example.

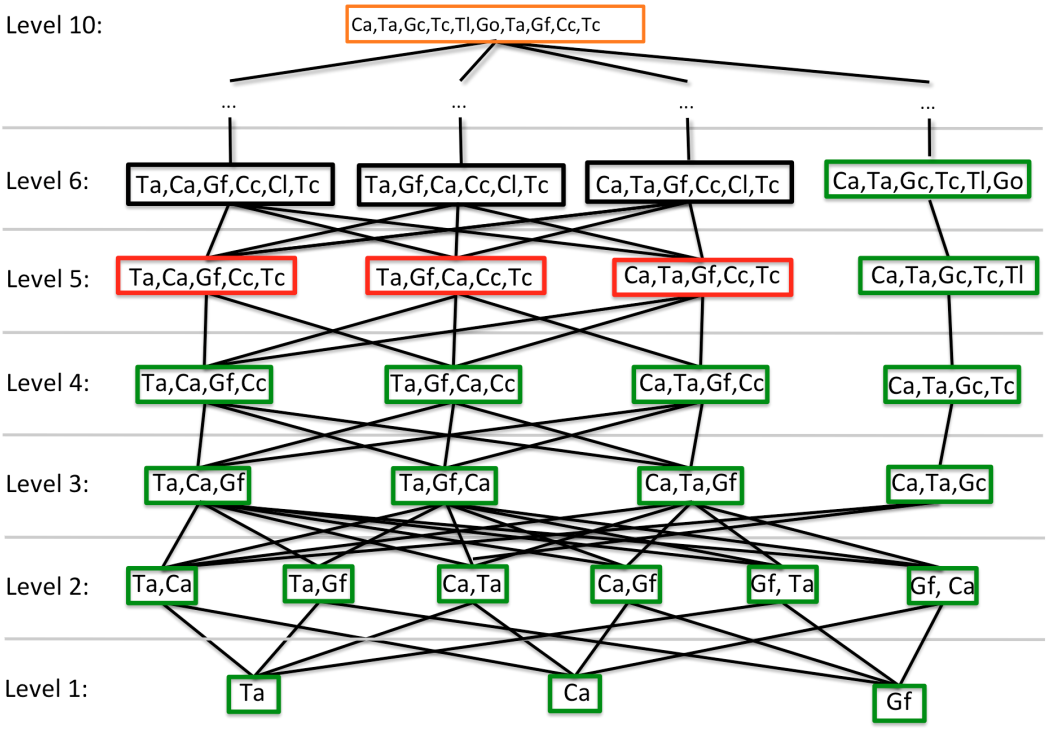


Figure 6.1: Subset-graph of the railroad crossing example.

- Black: a node is colored *black* if it represents a good execution trace, but at least one subset of the execution-trace on a level below is colored red. Black traces cannot be causal themselves, since they are good traces, but since a sub-trace of them with at least one element less 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 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 prevents the property violation.

- Orange: A node is colored *orange* if it represents a bad execution trace and at least one node on a 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 an 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". The trace "Ca,Ta,Gf,Cc,Tc" is a shorter red trace and a subset of the trace "Ca,Ta,Gc,Tc,Tl,Go,Ta,Gf,Cc,Tc", hence the trace does not fulfill the minimality constraint.

The pseudocode of the subset graph data structure is presented in Listing 6.1.

```
1  class SubSetGraph {
2
3  PrefixTree prefix_actions;
4  PrefixTree prefix_states;
5  List of Integers redTraces;
6
7  HashMap<Integer level, List of Traces> tracesToLevel;
8
9  //for duplicate state matching
10 HashMap<state, List of Traces> matchList;
11
12  function addTrace(Trace t) {
13     // ... see Listing 6.2
14  }
15
16  function checkAC22() {
17     // ... see Listing 6.3
18  }
19
20  function checkOC() {
21     // ... see Listing 6.4
22  }
23
24  function matchDuplicateState(State s, Trace t)
25  {
26     // ... see Listing 6.5
27  }
28
29  function searchFinished() {
30     checkAC22();
31     checkOC();
32  }
33 }
```

Listing 6.1: Algorithm sketch of the subset graph data structure.

The pseudocode for adding a trace is given in Listing 6.2. The algorithm first adds the actions and states of the trace to the corresponding prefix trees (Listing 6.2, lines 4-5), stores the trace ID in the list containing all traces (Listing 6.2, line 7), inserts all states of the trace into the match list (Listing 6.2, line 9) used for the duplicate state prefix matching described in Section 6.3.3, and adds the trace ID to the corresponding level of the subset graph (Listing 6.2, line 9). If the trace that is added is a bad trace, it is marked as red. If it is a good trace it is marked as green (Listing 6.2, lines 11-15). The algorithm then checks the subset relationships with the execution traces on the level below (level-1) (Listing 6.2, lines 17-30) and, if DFS is used, on the level above (level+1) as well (Listing 6.2, lines 32-48). It is not necessary to check the subset relationships on the level above (level+1) if BFS is used, because BFS adds the traces by increasing length and thus there are no traces

yet on the level above if BFS is used. If DFS is used it is possible that there already are traces on the level above and thus we have to check the subset relationships on the level above. Once all subset relationships are established, the nodes representing the executions are colored according to the above described coloring rules. If a trace is colored red, we additionally need to check whether there exists a shorter red trace more than one level below which is a subset of the new red trace (Listing 6.2, lines 50-63). It is possible that the shorter red trace ended in the property violating state and no orange or black trace exists as a super-trace of the shorter red trace. Consequently, no connection between the new red trace and the shorter red trace can be established through the subset graph. If such a shorter red trace is found, the current trace is colored orange. If DFS is used we also need to check whether a longer red trace exists that needs to be colored orange and removed from the redTraces list (Listing 6.2, lines 59-62).

The traces that are marked as red at the end of the addTrace method are added to the list of redTraces (Listing 6.2, lines 64-67). 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.

```
1  function addTrace(Trace t)
2  {
3    //add trace to the prefix trees
4    addTo(prefix_actions, t);
5    addTo(prefix_states, t);
6    //insert all states into the match list
7    addAllStatesToMatchList(t);
8    //add trace to level
9    addTraceToLevel(t.length, t);
10
11   IF(t.isBad()) {
12     t.color = red;
13   }ELSE{
14     t.color = green;
15   }
16
17   FOR EACH Trace t' in getTracesOnLevel(t.length-1) {
18     IF(t' is sub set of t) {
19         IF(t'.color = red)
20         {
21             IF(t.isBad()){
22                 t.color = orange;
23             }ELSE{
24                 t.color = black;
25                 addBlackSuperSet(t',t);
26                 addBlackSubSet(t,t');
27             }
28         }
29     }
30   }
```

```
31
32    IF(Algo = DFS)
33    {
34      FOR EACH Trace t' in getTracesOnLevel(t.length+1){
35          IF(t is sub set of t') {
36              IF(t.color = red)
37              {
38                  IF(t'.isBad()){
39                      t'.color = orange;
40                  }ELSE{
41                      t'.color = black;
42                      addBlackSuperSet(t,t');
43                      addBlackSubSet(t',t);
44                  }
45              }
46          }
47       }
48    }
49
50    FOR EACH Trace t' in redTraces {
51      IF(t' is sub set of t) {
52          IF(t.isBad()){
53              t.color = orange;
54          }ELSE{
55              t.color = black;
56              addBlackSuperSet(t',t);
57              addBlackSubSet(t,t');
58          }
59      }ELSE IF(ALGO = DFS & t is sub set of t'){
60          t'.color = orange;
61          removeFrom(redTraces, t');
62      }
63    }
64    IF(t.color = red)
65    {
66      addTo(redTraces, t);
67    }
68  }
```

Listing 6.2: Algorithm sketch of the addTrace method of the subset graph

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

Theorem 50. *AC1 is fulfilled for ψ_σ of each execution trace σ that is colored red.*

Proof. By definition an execution trace is only colored red if it is a bad trace and according to Theorem 45 AC1 is fulfilled for all $\sigma \in \Sigma_B$. □

Theorem 51. *AC2(1) is fulfilled for ψ_σ of each execution trace σ that is colored red.*

Proof. According to Theorem 46 we can show AC2(1) by finding an execution $\sigma' \in \Sigma_G$ for which $\sigma' \subset \sigma$ holds. For an execution σ to be colored red, all sub execution traces on the level below have to be colored green. Consequently, for each execution σ' for which $\sigma' \subset \sigma$ holds also $\sigma' \in \Sigma_G$ holds because it is colored green and hence needs to be a good trace. Therefore, AC2(1) is fulfilled according to Theorem 46. If σ has length one there exists an empty trace $\sigma' = \{\}$ which is a good trace and for which $\sigma' \vDash \sigma$ holds. Consequently, AC2(1) is fulfilled for ψ_σ of each execution trace σ that is colored red. □

Theorem 52. *If BFS is used, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red. If DFS is used, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red as soon as the state-space exploration has terminated.*

Proof. According to Theorem 49, ψ fulfills AC3 if there does not exist a trace $\sigma''' \in \Sigma_B$ for which $\sigma''' \subset \sigma$ holds. This is due to the fact that by definition an execution trace is only colored red if all its subsets are colored green, which means there is no bad sub-execution σ''' of σ. If BFS is used the shortest paths are added first, hence all sub-executions are known at the time where σ is inserted and colored. Consequently, if BFS is used, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red. If DFS is used it is possible that new sub-executions are found as long as the state-space exploration is not complete. As a result, AC3 is fulfilled for ψ_σ of each execution trace σ that is colored red as soon as the state-space exploration with DFS has terminated. □

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

According to Theorem 47, AC2(2) holds for ψ_σ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{\subset} \sigma''$ holds. If such a σ'' exists, it is a black superset of σ because $\sigma \subset \sigma''$ holds for each black superset of σ, and σ'' is only colored black if it is a good trace. Consequently, we need to check for each black superset σ'' of σ whether $\sigma \dot{\subset} \sigma''$ holds. If there is no σ'' for which $\sigma \dot{\subset} \sigma''$ holds, then ψ_σ fulfills AC2(2). If $\sigma \dot{\subset} \sigma''$ holds for a black superset, then we need to modify ψ_σ as specified by Definition 31. Hence, we have shown that AC1, AC2(1), AC2(2) and AC3 are fulfilled for ψ_σ of each red execution σ and, consequently, that ψ_σ is causal for the property violation.

The pseudocode for the AC2(2) test is shown in Listing 6.3. Notice that the AC2(2) test is needed in order to detect whether the non-occurrence of an event is causal. For each of the traces that are marked red, the minimal sets of events that can prevent the property violation on the red trace are computed (Listing 6.3, lines 1-40). This is achieved by comparing the red trace with all black superset traces (Listing 6.3, lines 6-23). For each black superset trace we check whether the set Q of event variables that conjunctively prevent the occurrence of the property violation is minimal and if that is the case add it to the list of Qs (Listing 6.3, lines 24-38). For the AC2(2) test it is necessary to store all traces that are colored black. We have added a runtime switch in the implementation of the causality checking method that allows the user to turn the AC2(2) test off in order to save memory at

the expense of not being able to take the possible causality of the non-occurrence of an event into account.

```
1  function checkAC22()
2  {
3   FOR EACH Trace t in redTraces
4   {
5     List Qs = {};
6     FOR EACH Traces t' in
7         getBlackSuperSetsFor(t)
8     {
9       IF(t is ordered subset of t'))
10      {
11        u = 0;
12        Q = {};
13        FOR x = 0 to sizeof(t'.actions)
14        {
15          IF(t.actions.get(u) = t'.actions.get(x)))
16          {//events are same move to next event
17            u++;
18          }
19          ELSE IF(t'.actions.get(x) NOT IN Q)
20          {
21            Q.add(t'.actions.get(x));
22          }
23        }
24        IF(Q != {})
25        {
26          add = true;
27          FOR EACH q in Qs
28          {
29              IF(Q is subset or equal q)
30              {
31                  add = false;
32              }
33          }
34          if(add)
35          {
36              Qs.add(Q);
37          }
38        }
39      }
40    }
41    t.addQs(Qs);
42  }
43 }
```

Listing 6.3: Algorithm sketch of the checkAC22 method.

If the AC2(2) test is fulfilled by ψ_σ, then the OC test is performed. The pseu-

docode for the method that performs the OC test is given in Listing 6.4. Due to the structure of the subset graph, it is sufficient for the OC test to check for each red execution trace whether there exists a red execution trace with the same length for which the unordered $\subseteq$ relationship holds (Listing 6.4, lines 1-29). 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 (Listing 6.4, lines 10-24). If a pair of events does not occur in the same order, then the order of this pair is marked as having no influence on causality, by adding an entry in the order constraint matrix. After the OC test is completed, the order constraint matrix contains all ordering information. The EOL formula ψ_σ representing all traces in this causality class is derived from the order constraint matrix. Finally all ψ_σ representing the different causality classes are added as disjuncts to the EOL formula Ψ. If there are duplicate ψ_σ from different interleavings of σ only one ψ_σ is added as a disjunct of Ψ (Listing 6.4, line 25).

```
1  function checkOC()
2  {
3
4   //order constraints are initialized
5   //with true before OC is executed
6   FOR EACH Trace t in redTraces
7   {
8       FOR EACH Trace t' in redTraces
9       {
10        IF(t.id != t'.id && t is equal t'){
11              FOR i = 0 to size(t.actions)
12              {
13                FOR i = j to size(t.actions)
14                {
15                  IF(!(
16                   t'.actions.indexOf(t.actions.get(i))
17                   <=
18                   t'.actions.indexOf(t.actions.get(j))
19                  ))
20                  {//pair i,j is not ordered
21                    t.OrderConstraints(i,j,false);
22                  }
23                }
24              }
25           remove(redTraces,t');
26         }
27      }
28    }
29 }
```

Listing 6.4: Algorithm sketch of the checkOC method.

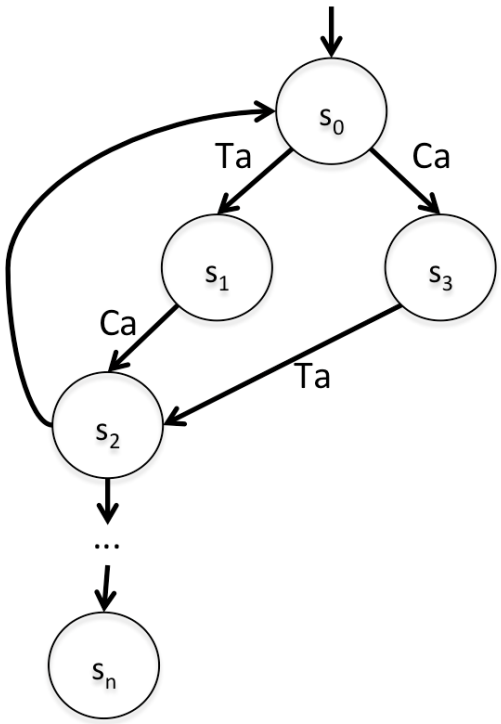


Figure 6.2: Partial state-space of the railroad crossing example.

6.3.3 Duplicate State Prefix Matching

The state-space exploration algorithms DFS (Listing 6.6) and BFS (Listing 6.7) store all already explored states in V. DFS stores the states that still have to be explored in the stack data structure S (Listing 6.6, line 3), while BFS stores the states that have to be explored in the queue data structure D (Listing 6.7, line 1). DFS and BFS terminate because during the exploration it is checked whether a state already is in V before it is explored (Listing 6.6, line 43 and Listing 6.7, line 33). If DFS or BFS encounter a state that is already in V its successors are not explored for a second time. If a state that already is stored in V is encountered by DFS or BFS a second time, we call this state a duplicate state.

Consider the partial state-space of the railroad crossing example shown in Figure 6.2, if DFS already has found the trace s_0 Ta s_1 Ca s_2 ... s_n and finds the new trace s_0 Ca s_3 Ta s_2 leading to the duplicate state s_2 already in V, the state s_2 is not further explored a second time, and the execution trace s_0 Ca s_3 Ta s_2 ... s_n is never found.

In Section 5.3 we required that all good and bad traces have been found in order for the causality checking to be complete and sound. If we would not take the traces that contain a duplicate state into account the causality checking result is potentially not complete and sound.

When DFS encounters a duplicate state, it is possible that the new trace to the duplicate state is shorter or has a different event order than the already known

execution traces leading to the duplicate state. Hence, the new execution trace is needed to ensure the AC3 condition and for the OC test to be able to detect all orderings. Note that it is also possible that the new trace is longer, for instance, the trace s_0 Ta s_1 Ca s_2 s_0 Ta s_1 Ca s_2 ... s_n that can be found in our example. Since for such a trace the minimality condition is violate it is not needed for the causality checking and hence is not added by the matchDuplicateState method.

BFS explores the state-space following an exploration order that leads to a monotonically increasing length of the execution traces. Consequently, the new execution trace found by BFS leading to the duplicate state either has the same length as the already known execution trace leading to the duplicate state, or the new execution trace is longer than the already known execution trace. If the new execution trace has the same length, the events on the trace have an order different from the one in the already known execution trace. Hence, the new execution trace is needed for the OC test to be able to detect all orderings. In order to ensure that all traces are generated when BFS is used, it is important that the matchDuplicateState method is called when the state-space exploration with BFS is finished. The reason for this is that when the trace s_0 Ca s_3 Ta s_2 leading to the duplicate state s_2 is found, only the trace s_0 Ta s_1 Ca s_2 is known but the trace s_0 Ta s_1 Ca s_2 ... s_n was not yet found and, consequently, the trace s_0 Ca s_3 Ta s_2 ... s_n can not be generated. If the trace to the duplicate state is stored in a list and the matchDuplicateState method is called once all other traces have been found, it can be ensured that all traces will be generated.

We have implemented a method called matchDuplicateState ensuring that all execution traces are found by replacing an old prefix leading to a duplicate state with the new prefix and adding the resulting trace to the sub-set graph. The prefix matching algorithm is shown in Listing 6.5.

```
1  function matchDuplicateState(State s, Trace t)
2  {
3      FOR EACH Trace t' in matchList.getTracesFor(s){
4          //replacing the state and events in t'
5          //leading to state s
6          //with the new state and events in t
7          //the states and events from s to the end of
8          //t' stay the same
9          Trace t'' = replacePrefix(s, t, t');
10         addTrace(t'');
11     }
12 }
```

Listing 6.5: Duplicate state prefix matching algorithm.

It is important to note that that the traces generated by the by the matchDuplicateState method are sound by definition, which means that no trace which is not possible in the system model can be generated by the matchDuplicateState method.

Theorem 53. *The execution traces generated by the matchDuplicateState method*

are sound by definition.

Proof. We already have found the trace $s_0\ a_{\alpha_1}\ s_1\ a_{\alpha_2}\ s_2\ ...\ s_n$ and find the new trace $s_0\ a_{\alpha_2}\ s_3\ a_{\alpha_1}\ s_2$ leading to the duplicate state s_2. Since we can reach s_n from s_2 and we have found a new execution trace leading to s_2 we can replace the old prefix $s_0\ a_{\alpha_1}\ s_1\ a_{\alpha_2}\ s_2$ with the new prefix $s_0\ a_{\alpha_2}\ s_3\ a_{\alpha_1}\ s_2$ and obtain the trace $s_0\ a_{\alpha_2}\ s_3\ a_{\alpha_1}\ s_2\ ...\ s_n$, which obviously is sound. Consequently, the traces generated by the matchDuplicateState method are sound. □

6.3.4 Integration into Depth-First Search

Listing 6.6 shows the pseudocode of the DFS algorithm in which we have integrated the causality checking approach. We adapted the DFS algorithm to add an execution trace to the subset graph data structure whenever either a bad state is reached (Listing 6.6, lines 13-20) or a good execution trace has been found (Listing 6.6, lines 21-28 and lines 32-39). If DFS is used it is sufficient to print the search stack in order to retrieve the execution trace. If a state is encountered that already is in the state-space the duplicate state prefix matching method described in Section 6.3.3 is called (Listing 6.6, lines 47-54).

```
1  SubSetGraph G = {};
2  State-space V = {};
3  Stack S = {};
4
5  function main(s)
6  {
7   dfs(init_state);
8   G.searchFinished();
9  }
10
11 function dfs(s)
12 {
13  IF(error(s))
14  {
15    trace = buildTrace(s);
16    trace.isBad = true;
17    G.addTrace(trace) ;
18    /*bad execution trace found,
19     add to causality computation*/
20  }
21  ELSE
22  {
23     trace = buildTraceFromStack(Stack);
24     trace.isBad = false;
25     G.addTrace(trace);
26     /*"so far good" execution trace found,
27     add to causality computation*/
28  }
29  add(V,s);
```

```
30  add(S,s);
31
32  if(hasNoSuccessors(s) & NOT error(s))
33  {
34    trace = buildTraceFromStack(Stack);
35    trace.isBad = false;
36    G.addTrace(trace);
37    /*good execution trace found,
38     add to causality computation*/
39  }
40
41  FOR EACH successor t of s
42  {
43    IF  in(V, t) == false
44    {
45      dfs(t)
46    }
47    ELSE
48    {
49          trace = buildTraceFromStack(Stack)
50          trace.isBad = false;
51          G.matchDuplicateState(t,trace);
52          /*Found new path to already known state,
53          add trace to match list*/
54    }
55  }
56 delete s from Stack
57 }
```

Listing 6.6: Algorithm sketch of the extended DFS algorithm.

6.3.5 Integration into Breadth-First Search

The pseudocode of the adapted BFS algorithm is given in Listing 6.7. When using breadth-first search, the execution trace leading from the initial state to a property violating state can be generated by an iterating backwards search through the predecessor links until an initial state is reached. Whenever a bad (Listing 6.7, lines 22-26) or a good execution (Listing 6.7, lines 26-30) is found, it is added to the subset graph. If a state is encountered that already is in the state-space the corresponding trace is added to the TracesToMatch list (Listing 6.7, lines 45-51) and once the state-space exploration is finished the duplicate state prefix matching method described in Section 6.3.3 is called for these traces (Listing 6.7, lines 11-13).

```
1  Queue D = {};
2  State-space V = {};
3  SubSetGraph G = {};
4  TracesToMatch = {};
5
6  function main(){
```

```
 7   add(V, init_state);
 8   add(D, init_state);
 9   bfs();
10
11   FOR EACH Pair State s, Trace t in TracesToMatch{
12      G.matchDuplicateState(s,trace);
13   }
14
15   G.searchFinished();
16  }
17
18  function bfs(){
19   //Returns top element of the queue and deletes it.
20   s = getHead(D);
21
22   IF(error(s)){ //bad trace found
23      trace = buildTrace(s);
24      trace.isBad = true;
25      G.addTrace(trace);
26   } ELSE {  //"so far good" trace found
27      trace = buildTrace(s);
28      trace.isBad = false;
29      G.addTrace(trace);
30   }
31
32   IF(hasNoSuccessors(s) & NOT error(s)){//good trace
33      trace = buildTrace(s);
34      trace.isBad = false;
35      G.addTrace(trace);
36   }
37
38   WHILE( s has successor t & G.searchNotCompleted ) {
39     IF in(V, t) == false { //generate traces
40       // (backwards linking)
41       setPrevious(t,s);
42       add(V,t);
43       add(D,t);
44       bfs();
45     } ELSE {
46         trace = buildTraceFromStack(Stack);
47         trace.isBad = false;
48         TracesToMatch.add(t,trace);
49         /*Found new trace to already known state t,
50         do prefix matching*/
51     }
52  }
```

Listing 6.7: Algorithm sketch of the adapted BFS algorithm.

6.3.6 Iterative Approach

In the standard version of the causality checking approach the bad and good traces found during state-space exploration are added to the sub-set graph and stored if necessary. Especially due to the storage of the potentially large number of good traces, the standard approach is not memory efficient. We will now propose an optimized version of the algorithm that consumes less memory. The iterative approach leverages the fact that the length of the bad traces found by BFS is monotonically increasing. The iterative approach is not implemented for DFS, because this would be memory inefficient.

The pseudocode of the adapted SubsetGraph data structure is shown in Listing 6.8 and the pseudocode of the adapted addTrace method for the iterative approach is shown in Listing 6.9. The SubsetGraph data structure is extended by two Boolean variables that indicate whether the algorithm is executing the first or the second iteration (Listing 6.8, lines 6-7). The functions implementing the AC2(2) and OC tests and the matchDuplicateState method do not have to be changed.

The iterative algorithm constitutes of two consecutively executed state-space explorations with BFS.

1. In the first state-space exploration, we limit the causality checking to find those causality classes that satisfy conditions AC1, AC2(1) and AC3. We define the root-node of the subset graph to be green and only add bad traces to the subset graph, these traces are only stored if they are colored red (Listing 6.9, lines 4-24).

2. In the second state-space exploration, we focus on finding the causal event orderings of the previously identified causality classes and check the AC2(2) test for the previously identified causality classes (Listing 6.9, lines 26-64).

The pseudocode for the adapted BFS algorithm is shown in Listing 6.10, where BFS is executed in two consecutive iterations (Listing 6.10, lines 7-18). In the first iteration only bad traces are added to the SubsetGraph data structure (Listing 6.10, lines 31-35), while in the second iteration also the good traces (Listing 6.10, lines 35-49) and the traces generated by the matchDuplicateState method (Listing 6.10, lines 20-22 and lines 58-66) are added to the SubsetGraph data structure.

As we will show in the experimental evaluation in Section 6.6 the iterative approach leads to a reduction of the memory consumption in comparisons to the memory consumed by the standard causality checking approach.

```
1  class SubSetGraph {
2
3  PrefixTree prefix_actions;
4  PrefixTree prefix_states;
5  List of Integers redTraces;
6  Boolean isFirstRun = false;
7  Boolean isSecondRun = false;
8
9  HashMap<Integer level, List of Traces> tracesToLevel;
10
11 //for duplicate state matching
12 HashMap<state, List of Traces> matchList;
13
14  function addTrace(Trace t) {
15     // ... (*see Listing 6.9*)
16  }
17
18  function checkAC22() {
19     // ... (*see Listing 6.3*)
20  }
21
22  function checkOC() {
23     // ... (*see Listing 6.4*)
24  }
25
26  function matchDuplicateState(State s, Trace t)
27  {
28     // ... (*see Listing 6.5*)
29  }
30
31  function searchFinished() {
32     checkAC22();
33     checkOC();
34  }
35 }
```

Listing 6.8: Algorithm sketch of the subset graph data structure for the iterative causality checking approach.

```
1  function addTrace(Trace t)
2  {
3
4    IF(isFirstRun && t.isBad())
5    {
6      FOR EACH Trace t' in redTraces {
7          IF(t' is sub set of t) {
8              t.color = orange;
9          }
10     }ELSE IF(t is sub set of t'){
11         t'.color = orange;
12     }
13     IF(t.color = red)
14     {
15         addTo(redTraces, t);
16         //add trace to the prefix trees
17         addTo(prefix_actions, t);
18         addTo(prefix_states, t);
19         //insert all states into the match list
20         addAllStatesToMatchList(t);
21         //add trace to level
22         addTraceToLevel(t.length, t);
23     }
24   }
25
26   IF(isSecondRun)
27   {
28     IF(t.isBad()) {
29         t.color = red;
30     }ELSE{
31         t.color = green;
32     }
33
34     FOR EACH Trace t' in redTraces {
35         IF(t' is sub set of t) {
36             IF(t.isBad()){
37                 t.color = orange;
38             }ELSE{
39                 t.color = black;
40                 addBlackSuperSet(t',t);
41                 addBlackSubSet(t,t');
42             }
43         }ELSE IF(t is sub set of t'){
44             t'.color = orange;
45         }
46     }
47     IF(t.color = red)
48     {
49         addTo(redTraces, t);
```

```
50      }
51
52      IF(t.color = red | t.color = black)
53      {
54          //add trace to the prefix trees
55          addTo(prefix_actions, t);
56          addTo(prefix_states, t);
57          //add to list with all traces
58          addTo(allTraces, t);
59          //insert all states into the match list
60          addAllStatesToMatchList(t);
61          //add trace to level
62          addTraceToLevel(t.length, t);
63      }
64    }
65
66  }
```

Listing 6.9: Algorithm sketch of the addTrace method for the iterative causality checking approach.

```
1  Queue D = {};
2  State-space V = {};
3  SubSetGraph G = {};
4  TracesToMatch = {};
5
6  function main(){
7   //first run
8   G.isFirstRun = true;
9   add(V, init_state);
10  add(D, init_state);
11  bfs();
12
13  //reset and do second run
14  G.isFirstRun = false;
15  G.isSecondRun = true;
16  add(V, init_state);
17  add(D, init_state);
18  bfs();
19
20  FOR EACH Pair State s, Trace t in TracesToMatch{
21     G.matchDuplicateState(s,trace);
22  }
23
24  G.searchFinished();
25 }
26
27 function bfs(){
28  //Returns top element of the queue and deletes it.
29  s = getHead(D);
```

```
30
31  IF(error(s)){ //bad trace found
32     trace = buildTrace(s);
33     trace.isBad = true;
34     G.addTrace(trace);
35  } ELSE {  //"so far good" trace found
36     IF(G.isSecondRun){
37         trace = buildTrace(s);
38         trace.isBad = false;
39         G.addTrace(trace);
40     }
41  }
42
43  IF(hasNoSuccessors(s) & NOT error(s)){//good trace
44     IF(G.isSecondRun){
45         trace = buildTrace(s);
46         trace.isBad = false;
47         G.addTrace(trace);
48     }
49  }
50
51  WHILE( s has successor t & G.searchNotCompleted ) {
52    IF in(V, t) == false { //generate traces
53      // (backwards linking)
54      setPrevious(t,s);
55      add(V,t);
56      add(D,t);
57      bfs();
58    } ELSE {
59       IF(G.isSecondRun){
60        trace = buildTraceFromStack(Stack);
61        trace.isBad = false;
62        TracesToMatch.add(t,trace);
63        /*Found new trace to already known state t,
64        do prefix matching*/
65       }
66    }
67 }
```

Listing 6.10: Algorithm sketch of the adapted BFS algorithm for the iterative causality checking approach.

6.3.7 Iterative Approach with Parallel Breadth-First Search

As we will see in Section 6.6, BFS outperforms DFS in terms of runtime and memory consumption. In order to further optimize the runtime we have extended the parallel BFS variant already implemented in the SpinJa [33] model-checker to support causality checking.

There are three parallelization strategies for DFS and BFS that are discussed in the literature [10, 11]:

1. Static partitioning of the state space, where each thread maintains its own queue and store for the already visited states. This parallelization variant is limited to BFS.

2. Stack slicing, where a shared store for the already visited states is used and each thread maintains its own search stack and the algorithm tries to balance the load by shifting states from the stack of one search thread to stack of another search stack. This parallelization variant is limited to DFS.

3. Shared storage, where the store for the already visited states and the queue is shared by all threads.

The adapted parallel BFS algorithm is shown in Listing 6.11. The parallelization of the BFS algorithm is achieved by executing a predefined number (maxThreads) of BFS threads (Listing 6.11, lines 12-14) with a shared queue, state-space, and subset graph. This form of parallelization was shown to be efficient for multi-core systems by Laarman et al. [67]. We choose a shared storage parallelization instead of static partitioning of the state-space because of the high communication costs that are caused by a static partitioning and because the causality checking requires a shared subset graph data-structure. Stack slicing could not be used because it is limited to DFS. Each parallel BFS thread retrieves a state from the shared queue and adds the successor states to the shared queue and checks whether the property is violated in one of the successor states (Listing 6.11, lines 33-73). For the iterative approach with parallel BFS we do not need to change the subset graph implementation and can use the subset graph implementation of the iterative approach proposed in Section 6.3.6.

```
1  Queue D = {};
2  State-space V = {};
3  SubSetGraph G = {};
4  TracesToMatch = {};
5
6  function main(){
7   //first run
8   G.isFirstRun = true;
9   add(V, init_state);
10  add(D, init_state);
11
```

```
12  FOR i=0 TO maxThreads {
13    NewThread(bfs());
14  }
15
16  //reset and do second run
17  G.isFirstRun = false;
18  G.isSecondRun = true;
19  add(V, init_state);
20  add(D, init_state);
21
22  FOR i=0 TO maxThreads {
23    NewThread(bfs());
24  }
25
26  FOR EACH Pair State s, Trace t in TracesToMatch{
27     G.matchDuplicateState(s,trace);
28  }
29
30  G.searchFinished();
31 }
32
33 function bfs(){
34  //Returns top element of the queue and deletes it.
35  s = getHead(D);
36
37  IF(error(s)){ //bad trace found
38     trace = buildTrace(s);
39     trace.isBad = true;
40     G.addTrace(trace);
41  } ELSE {  //"so far good" trace found
42     IF(G.isSecondRun){
43         trace = buildTrace(s);
44         trace.isBad = false;
45         G.addTrace(trace);
46     }
47  }
48
49  IF(hasNoSuccessors(s) & NOT error(s)){//good trace
50     IF(G.isSecondRun){
51         trace = buildTrace(s);
52         trace.isBad = false;
53         G.addTrace(trace);
54     }
55  }
56
57  WHILE( s has successor t & G.searchNotCompleted ) {
58    IF in(V, t) == false { //generate traces
59      // (backwards linking)
60      setPrevious(t,s);
```

```
61        add(V,t);
62        add(D,t);
63        bfs();
64      } ELSE {
65         IF(G.isSecondRun){
66          trace = buildTraceFromStack(Stack);
67          trace.isBad = false;
68          TracesToMatch.add(t,trace);
69          /*Found new trace to already known state t,
70          do prefix matching*/
71         }
72      }
73  }
```

Listing 6.11: Algorithm sketch of the adapted parallel BFS algorithm for the iterative causality checking approach.

6.4 Completeness and Soundness

In this section we discuss how the general results for completeness and soundness of causality checking presented in Section 5.3 can be used to show completeness and soundness of the qualitative causality checking approach. In the remainder of this section we will assume that the state-space of the model we analyze is finite and a complete exploration of the state-space is possible. In Section 9.2 we will discuss the implications if this assumption does not hold any longer.

6.4.1 Completeness

According to Theorem 43, the event order logic formula returned by the causality checker is complete if all bad traces have been identified.

Therefore, we have to show that the DFS and BFS algorithms on which the causality checking algorithm is based completely identify all bad traces.

Theorem 54. *If the state-space of the model we analyze is finite and a complete exploration of the state-space is possible, the adapted DFS and BFS algorithms identify all bad traces and thus the causality checking algorithm is complete.*

Proof. If the state-space of the model is finite, then the DFS and BFS algorithms used for model checking are able to perform a full state-space exploration and identify all possible bad states [8]. If the DFS algorithm or the BFS algorithm encounter a state that was previously explored and hence all successors of this state have already been explored as well, the successors are not explored for a second time. In order to get the complete set of traces we use the duplicate state matching method discussed in Section 6.3.3. Since all bad states have been identified, there exists at least one trace that leads to each bad state. For each duplicate state on this trace the duplicate state matching method will generate all possible traces

that lead from the initial state to the duplicate state and finally to the bad state. Consequently, we can guarantee that all bad traces have been identified and thus the result of the causality checking algorithm is complete. □

6.4.2 Soundness

Theorem 44 makes two assumptions in order to show the soundness of the causality checking result.

1. It is assumed that the bad traces used for the causality checking are sound. The bad traces returned by the DFS and BFS algorithms on which we base our causality checking algorithm are sound by definition and according to Theorem 53 the traces generated by the duplicate state matching method are sound.

2. It is assumed that all good traces have been found. If the state-space of the model is finite all states in the state-space are explored at least once. Consequently, for each good state there exists at least one trace leading to this state. For each duplicate state on this trace the duplicate state matching method will generate the all possible traces that lead from the initial state to the duplicate state and finally to the good state. Consequently, all good traces are identified.

Corollary 3. *It follows that if the state-space of the model we analyze is finite and a complete exploration of the state-space is possible the results generated by the qualitative causality checking approach are sound.*

6.5 Complexity Considerations

Eiter and Lukasiewicz present in [39] a careful analysis of the complexity of computing causality with the actual cause definition of Halpern and Pearl in structural equation models. Most notable is the result that even for a structural equation model with only binary variables, in the general case computing causal relationships between variables is NP-complete. The reason for this is the fact that the event variables in X, Z and W are seen as independent and thus arbitrary combinations of their values have to be considered.

Results in [40] show that causality can be computed in polynomial time if the causal model M is domain-bounded and the causal graph over the events forms a directed causal tree. A directed causal tree consists of directed paths, where the nodes represent events, and the edges represent the causality relationships and the root node represents the hazard or effect. Intuitively this restriction means that there are know cyclic dependencies between the variables and that there exists a dependency between the variables in X, Z and W. In the system model we analyze there exists a dependency between all event variables, since each execution trace of the system model is a directed path containing the variables representing the

events. Consequently, computing the causal events for a property violation in a system model can be done in polynomial time.

For the qualitative causality checking approach proposed in Section 6.3, adding a trace (addTrace) to the subset graph has a worst-case runtime complexity of RT(addTrace)$\in \mathcal{O}(|t|)$ where $|t|$ is the number of traces. The test for AC2(2) (checkAC22) and OC (checkOC) have a worst-case runtime complexity of $\mathcal{O}(|t|^2)$, respectively. The function addTrace is called for each trace, and the function checkAC22 and checkOC are called once the search is finished. Thus we get a worst-case runtime complexity of RT(CausalityChecking) = $|t| * |t| + |t|^2 + |t|^2 \in \mathcal{O}(|t|^2)$ for the qualitative causality checking approach, where $|t|$ is the number of traces. Note that this is the runtime complexity added by the causality checking on top of the runtime of the state-space exploration. In the worst-case $|t| = 2^{|E|}$ where $|E|$ is the number of transitions in the model.

The worst-case runtime complexity of the iterative approach described in Section 6.3.6 as well as the runtime complexity of the iterative approach with parallel BFS described in Section 6.3.7 is in the worst-case the same as for the standard qualitative causality checking approach, since in the worst-case all traces could be bad traces. But in practice only a small number of traces is marked as red, hence the runtime in practical application scenarios is reduced with the iterative approaches.

In terms of memory consumption the worst-case for all approaches is that all traces have to be stored, consequently, the worst-case memory consumption is in $\mathcal{O}(|t|)$ where $|t|$ is the number of traces.

6.6 Experimental Evaluation

In order to evaluate the proposed approach, we have implemented our causality checking algorithms within the SpinJa toolset [33], a Java re-implementation of the explicit state model checker Spin [50]. Our SpinCause tool computes the causality relationships for a Promela model and a given LTL property. In order to compute all interleavings and all executions, partial order reduction was disabled during the state-space exploration. The Promela models used for the case studies have been created manually. In practical usage scenarios the Promela models can also be automatically synthesized from higher-level design models, as for instance by the QuantUM tool [72]. The following experiments were performed on a PC with two Intel Xeon Processors (4 cores with 3.60 Ghz) and 144 GBs of RAM.

We evaluate the causality checking approach using five case studies, which we discuss in Sections 6.6.1-6.6.5. In Section 6.6.6 a summary of the number of states, transitions, bad states and bad traces of the different case studies is given. Furthermore, we compare in Section 6.6.6 the runtime and memory consumption needed for a full state-space exploration of the case studies with DFS and BFS, with the runtime and memory consumption of the standard qualitative causality checking approach, the iterative approach and the iterative approach with parallel BFS, and summarize our results.

6.6.1 Railroad Crossing

The Promela model of the railroad crossing that we introduced in Section 3.2 comprises 133 states and 237 transitions. A total of 15 bad states and 41 bad execution traces are found. We want to compute the causal events for the hazardous case where both the car and the train are on the railroad crossing at the same time. The non-reachability of this hazard can be characterized by the LTL formula $\varphi = \Box\neg(car\_crossing \wedge train\_crossing)$. The causality checking algorithm generates the event order logic formula $\Psi = (\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_{<} \neg\mathrm{Cl} \barwedge_{>} \mathrm{Tc})) \vee ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_{<} \neg\mathrm{Cl} \barwedge_{>} (\mathrm{Gc} \wedge \mathrm{Tc}))$, consisting of two causality classes, for the violation of φ.

- First, if the gate fails at some point of the execution and a train (Ta) and a car (Ca) are approaching this results in a hazardous situation if the car is on the crossing (Cc) and does not leave the crossing (Cl) before the train (Tc) enters the crossing, which is described by the EOL formula $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_{<} \neg\mathrm{Cl} \barwedge_{>} \mathrm{Tc}))$.
- Second, if a train (Ta) and a car (Ca) are approaching but the gate closes (Gc) when the car (Cc) is already on the railway crossing and is not able to leave (Cl) before the gate is closing and the train is crossing (Tc), this also corresponds to a hazardous situation, and is described by the EOL formula $(\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_{<} \neg\mathrm{Cl} \barwedge_{>} (\mathrm{Gc} \wedge \mathrm{Tc})$.

6.6.2 Airbag System

The industrial size model of an airbag system that we use in this case study is taken from [2]. The architecture of this system, schematically shown in Figure 6.3, was provided by TRW Automotive GmbH.

The airbag system can be divided into three major parts: sensors, crash evaluation and actuators. The system we consider here consists of two acceleration sensors whose task it is to detect front or rear crashes, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag. Notice that the redundant acceleration sensors are mounted into different directions so that one is measuring the acceleration in the front direction of the vehicle and the other one is measuring the acceleration in the rear direction. The deployment of the airbag is also secured by two redundant protection mechanisms. The Field Effect Transistor (FET) controls the power supply for the airbag squibs that ignite the airbag. If the Field Effect Transistor is not armed, which means that the FET-pin is not high, the airbag squib does not have enough electrical power to ignite the airbag. The second protection mechanism is the Firing Application Specific Integrated Circuit (FASIC) which controls the airbag squib. Only if it receives first an arm command and then a fire command from the microcontroller it will ignite the airbag squib which leads to the pyrotechnical detonation inflating the airbag.

Although airbags save lives in crash situations, they may cause fatal accidents if they are inadvertently deployed. This is because the driver may lose control of

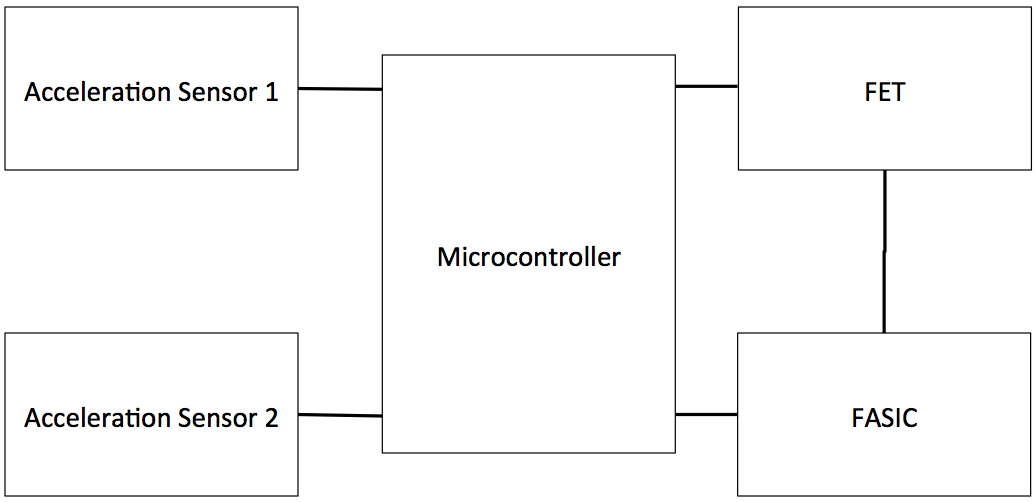


Figure 6.3: Block diagram showing the architecture of the airbag system.

the car when an inadvertently deployment of the airbag occurs. It is a pivotal safety requirement that an airbag is never deployed if there is no crash situation. We are interested in computing the causal events for the hazard corresponding to an inadvertent ignition of the airbag. The non-reachability of this hazard can be characterized by the LTL formula $\varphi = \Box\neg(no\_crash \wedge airbag\_deployed)$. The Promela model of the airbag system consists of 2,952 states and 25,340 transitions.

While there are a total of 912 bad states and 1399 bad execution traces, the causality checker result comprises only 5 causality classes. Obviously, a manual analysis of this large number of traces in order to determine causal factors would be very laborious.

The event order logic formula returned by the causality checker is Ψ = (FASICShortage) $\vee$ (FETStuckHigh $\wedge$ FASICStuckHigh) $\vee$ (MicroControllerFailure $\wedge$ enableFET $\wedge$ armFASIC $\wedge$ fireFASIC) $\vee$ (FETStuckHigh $\wedge$ MicroControllerFailure $\wedge$ armFASIC $\wedge$ fireFASIC) $\vee$ (MicroControllerFailure $\wedge$ enableFET $\wedge$ FASICStuckHigh).

- The *FASICShortage* event is the only event that can directly cause an inadvertent deployment and is represented by the event order logic formula (FASICShortage). The *FASICShortage* event models the case where there is an electrical short circuit in the FASIC component that directly leads to the deployment of the airbag.

- The combination of the events *FETStuckHigh* and *FASICStuckHigh* leads to an inadvertent deployment of the airbag if the event *FETStuckHigh* occurs prior to the event *FASICStuckHigh*, which is represented by the event order logic formula (FETStuckHigh $\wedge$ FASICStuckHigh). The event *FETStuckHigh* models a component level failure of the FET where the output pin of the FET is stuck at the value high, similarly *FASICStuckHigh* models a component level failure of the FASIC where the output port of the FASIC is stuck at

the value high. Since the electrical power that is needed to ignite the airbag is controlled by the FET, the *FETStuckHigh* event has to happen before the *FASICStuckHigh* event occurs, because otherwise there is not enough power available to ignite the airbag when the *FASICStuckHigh* event occurs.

- The event *MicroControllerFailure* can lead to an inadvertent deployment if it is followed by the following sequence of events: *enableFET*, *armFASIC*, and *fireFASIC*. This sequence is represented by the event order logic formula (MicroControllerFailure ∧ enableFET ∧ armFASIC ∧ fireFASIC). This causality class represents the scenario where through a failure of the program counter of the microcontroller the firing sequence *enableFET*, *armFASIC*, and *fireFASIC* is inadvertently executed.
- If the event *FETStuckHigh* occurs prior to the *MicroControllerFailure* event the sequence in which *armFASIC* and *fireFASIC* occur after the *MicroControllerFailure* event suffices to lead to an inadvertent deployment of the airbag. This sequence is represented by the event order logic formula (FETStuckHigh ∧ MicroControllerFailure ∧ armFASIC ∧ fireFASIC).
- If the event *FASICStuckHigh* occurs after the *MicroControllerFailure* event and the *enableFET* event, this also leads to an inadvertent deployment. It is represented by the event order logic formula (MicroControllerFailure ∧ enableFET ∧ FASICStuckHigh).

From the EOL formula returned by the causality checker it is easy to see that the ordering of the events, like for instance the order of the events (FETStuckHigh and FASICStuckHigh), is causal for the property violation. If traditional model checking and manual counterexample analysis is used, the same insight requires the comparison of the order of all events in all 1399 bad traces which is a time consuming and error prone task.

6.6.3 Embedded Control System

The embedded control system taken from [63] is part of the PRISM benchmark suite [66]. The system consists of a main processor, an input processor, an output processor, three sensors, two actuators and a communication bus connecting the processors. The architecture of the embedded control system is schematically displayed in Figure 6.4.

The input processor reads and processes the data provided by the three sensors, the main processor polls the input processor and forwards the data from the input processor to the output processor. The output processor controls the two actuators. Any of the three sensors can fail, but since they are used in triple modular redundancy the input processor can determine sufficient information if two of the three are functional. If more than one sensor fails the input processor reports the failure to the main processor and the main processor shuts the system down. Similarly, it is sufficient if at least one of the actuators is operational. If both actuators fail,

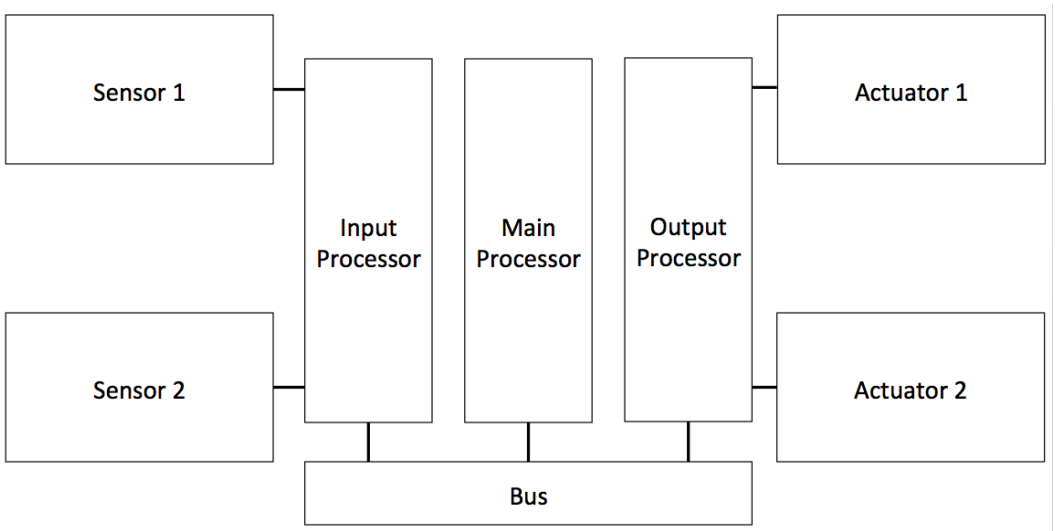


Figure 6.4: Block diagram showing the architecture of the embedded control system.

the output processor will report this two the main processor and the main processor shuts the system down. The input, output and main processor can also fail and this will also result in a shutdown of the system. We set the model constant MAX_COUNT, which represents the maximum number of processing failures that are tolerated by the main processor, to a value of 5. We are interested in computing the causal events for the event "system shut down". The non-reachability of this hazard can be characterized by the LTL formula $\varphi = \Box\neg(down)$.

We manually translate the PRISM model to a Promela model which comprises 6,013 states and 25,340 transitions and contains a total of 83 bad states and 90 bad execution traces.

The event order logic formula returned by the causality checker is Ψ = MainProcFail $\vee$ InputProcFail $\vee$ OutputProcFail $\vee$ InputProcTransFail $\vee$ (SensorFailure $\wedge$ SensorFailure) $\vee$ (ActuatorFailure $\wedge$ ActuatorFailure) and illustrates that a system shut down can be caused by

- a failure in the main processor (MainProcFail),
- a failure in the input/output processor (Input/OutputProcFail),
- a transient failure in the input processor (InputProcTransFail), or
- the failing of at least two sensors / actuators (SensorFailure and SensorFailure/ ActuatorFailure and ActuatorFailure).

6.6.4 Train Odometer Controller

The train odometer system taken from [22] consists of two independent sensors, a wheel sensor and a radar sensor, used to measure the speed and the position of a train. A monitor component continuously checks the status of both sensors.

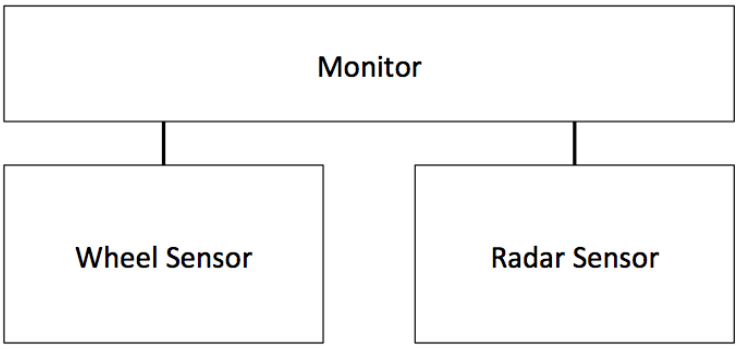


Figure 6.5: Block diagram showing the architecture of the train odometer controller.

The monitor reports failures of the sensors to other train components that have to disregard temporarily erroneous sensor data. If both sensors fail, the monitor initiates an emergency brake maneuver and the system is brought into a safe state. Only if the monitor fails, any subsequent faults in the sensors will no longer be detected. We are interested in computing the causal events for reaching an unsafe state of the system, where the failure of the sensors is not detected. The non-reachability of this hazard can be characterized by the LTL formula $\varphi = \Box\neg(failure\_not\_detected)$. The Promela model of the train odometer comprises 11,722 states, 14,049 transition, 1368 bad states, and 1579 bad execution traces.

The event order logic formula returned by the causality checker is Ψ = (Start_W_Fail_S $\wedge$ Wait_W_Fail_S) $\wedge$ (¬failureDeteted $\wedge_]$ Wait_Mon_Fail) $\vee$ (Start_W_Fail_F $\wedge$ Wait_W_Fail_F) $\wedge$ (¬failureDeteted $\wedge_]$ Wait_Mon_Fail) $\vee$ Wait_R_Fail $\wedge$ (¬failureDeteted $\wedge_]$ Wait_Mon_Fail) and illustrates that an unsafe state of the system is caused if

- the wheel sensor fails at low speed (Start_W_Fail_S $\wedge$ Wait_W_Fail_S), or high speed (Start_W_Fail_F $\wedge$ Wait_W_Fail_F), and the monitor fails prior to detecting a failure (¬failureDeteted $\wedge_]$ Wait_Mon_Fail), of

- the radar sensor fails (Wait_R_Fail) and the monitor fails prior to detecting a failure (¬failureDeteted $\wedge_]$ Wait_Mon_Fail).

From the EOL formula returned by the causality checker it is easy to see that the non-occurrence of the event failure detected ((¬failureDeteted) is causal. It is difficult to conclude this from the counterexamples generated by a model checker, since it also requires to compare the counterexamples with good execution traces. Consequently, causality checking provides an insight into the events causing the property violation which would not have been obvious using traditional model checking and manual counterexample analysis.

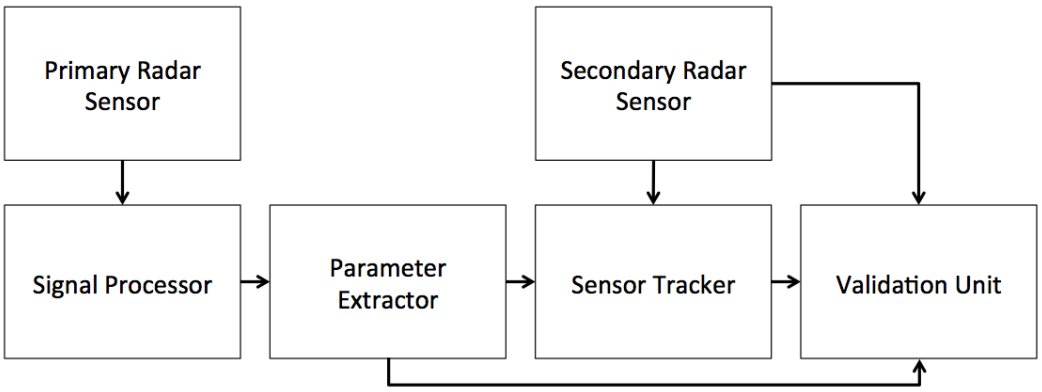


Figure 6.6: Block diagram showing one of the processing channels of the airport surveillance radar system.

6.6.5 Airport Surveillance Radar

The Airport Surveillance Radar (ASR) is developed at Airbus Defence & Space[1]. The high-level architecture of the physical components of the ASR system together with high-level behavior of the embedded software were specified in joint work with Beer et al. as a model in the SysML and first presented in [13].

The ASR monitors the airspace in the vicinity of an airport. It is used by air traffic controllers guiding aircraft according to their flight plan.

The architecture of the ASR system, which is schematically depicted in Figure 6.6, consists of three main components:

1. The Primary Surveillance Radar (PSR) which uses radar signals that are reflected from the body of an object moving in the airspace in order to determine its location. In the normal case this object is an aircraft.

2. The Secondary Surveillance Radar (SSR) which communicates with the transponder of the aircraft in order to obtain, amongst others, location and identity information.

3. The internal structure of the radar consists of two channels. Each channel processes the data provided by the PSR and the SSR and creates tracks representing the flight paths of the aircrafts. A channel consists of 4 components:

   (a) A signal processor which converts the analog PSR signal into digital signals.

   (b) A parameter extractor that is extracting plots from the digital signal. It is also managing a clutter-map which indicates the position of object irrelevant for the air traffic.

   (c) A sensor tracker which takes the SSR signal and the plots generated by the parameter extractor and combines them into a track representing

[1] `http://www.airbusdefenceandspace.com/`

the flightpath of an aircraft. In case of an error with the SSR or PSR signal the sensor tracker tries to extrapolate the track based on the given information. If the track cannot be recovered into its original state it is removed from the list.

(d) A *validation unit* that checks whether the tracks generated by the sensor tracker are plausible or not.

The tracks generated by the processing channel for each aircraft are displayed to the air traffic controller. Even though a complete failure of the system is highly critical for the safety of the system, the displaying of wrong data to the air traffic controller is considered to be the most critical hazard that can occur. Therefore, we compute the causal events for the case where the positional information of an aircraft is lost. The state that aircraft data for one aircraft is no longer displayed correctly is called *coasted track*, consequently, this hazard can be characterized by the LTL formula $\varphi = \Box\neg(coastedTrack)$.

We analyze two variants of the ASR system, the first variant where we have modeled only one channel, and the second variant where we have modeled both redundant channels. The one channel variant comprises 1,230,516 states and 7,492,866 transitions and the redundant two channel variant comprises 46,389,412 states and 326,412,170 transitions. The one channel variant contains 608,256 bad states and 611,376 bad traces and the two channel variant contains 15,206,400 bad states and 15,285,473 bad traces.

The event order logic formula returned by the causality checker is Ψ = (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ PSR_only $\wedge$ coastTrack) $\vee$ (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ SSR_only $\wedge$ coastTrack) and illustrates that whenever the following events occur in this order

1. the radar signal of an aircraft is processed by the signal processor (dot_sighted),
2. the parameter extractor detects a radar plot (plot_detected) representing the aircraft,
3. a track representing the flight path of the aircraft is initialized by the sensor tracker,
4. the primary radar (PSR_only) or the secondary radar (SSR_only) fails, and
5. the decision to no longer display the aircraft (coastTrack) is made

the flight track information of the aircraft is no longer visible for the air traffic controller.

Obviously, in an industrial setting, with time and cost constraints, it is not feasible to manually analyze all 15,285,473 bad traces of the ASR 2 channel variant in order to derive the causal events manually. Consequently, causality checking provides valuable information as to why the hazard occurred, which is very tedious or

even impossible to determine if standard model checking and manual counterexample analysis is used.

6.6.6 Discussion

Table 6.1 compares the number of states, transitions, bad states, and bad transitions for the different cases studies and whether the case study is an academic or an industrial case study.

	Number of					Industrial
	State	Transitions	Bad States	Bad Traces	Causality Classes	Case Study
Railroad	133	237	15	41	2	no
Airbag	2,952	25,340	912	1399	5	yes
Embedded	6,013	25,340	83	90	4	no
Train Odometer	11,722	14,049	1368	1579	2	no
ASR 1 Channel	1,230,516	7,492,866	608,256	611,376	2	yes
ASR 2 Channel	46,389,412	326,412,170	15,206,400	15,285,473	2	yes

Table 6.1: Number of states, transitions and bad states of the different case studies.

From Table 6.1 and the results obtained by the causality checking we can make the following observations:

- The number of causality classes computed by the causality checker is significantly less than the number of bad traces returned by the causality checker and, consequently, the results obtained by the causality checker are much easier to interpret as the counterexamples generated by the model checker.
- From the EOL formula returned by the causality checker it is easy to see that the ordering of the events, like for instance the order of the events (FETStuckHigh and FASICStuckHigh) in the airbag case study, is causal for the property violation. If traditional model checking and manual counterexample analysis is used, the same insight requires the comparison of the order of all events in all bad traces which is, due to the potentially large number of bad traces, a time consuming and error prone task.
- From the EOL formula returned by the causality checker it is easy to see that the non-occurrence of an event, as for instance the event failure detected (¬failureDeteted) in the train odometer case study, is causal. It is difficult to conclude this from the counterexamples generated by a model checker, since it also requires to compare the counterexamples with good execution traces. Consequently, causality checking provides an insight which would not have been obvious using traditional model checking and manual counterexample analysis.

We will now compare and discuss the runtime and memory consumption needed for a full state-space exploration of the causality checking with DFS and BFS, with

the runtime and memory consumption of the standard qualitative causality checking approach, the iterative approach and the iterative approach with parallel BFS. The

	Model Checking			
	DFS		BFS	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
Railroad	0.02	8.87	0.02	8.69
Airbag	0.17	9.23	0.18	9.06
Embedded	0.05	8.94	0.06	8.76
Train Odometer	0.26	9.79	0.27	9.62
ASR 1 Channel	44.95	8,467.14	51,66	8,466.88
ASR 2 Channel	503.92	14,669.11	659.26	14,668.85
	Causality Checking			
	DFS		BFS	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
Railroad	0.06	18.25	0.10	8.97
Airbag	0.86	165.52	1.24	21.19
Embedded	0.13	19.95	0.16	9.43
Train Odometer	15.06	2280.86	2.59	63.36
ASR 1 Channel	out of mem.	out of mem.	750,31	24,663.61
ASR 2 Channel	out of mem.	out of mem.	out of mem.	out of mem.

Table 6.2: Runtime and memory needed for model checking of the case studies with DFS and BFS and for causality checking with DFS and BFS.

runtime and memory needed for model checking of the different case studies with DFS and BFS and the runtime and memory needed for causality checking including model checking with DFS and BFS is given in Table 6.2. The different runtime values are visualized in Figure 6.7 and the memory consumption is visualized in Figure 6.8. The runtime and memory consumption for causality checking of the ASR case study with one and two channels using DFS and for the ASR case study with two channels using BFS can not be given because the algorithm consumed the complete 144 GBs of available memory and produced an out of memory error (out of mem.) prior to the completion of the causality checking.

The following trends can be identified:

- If no causality checking is done, DFS and BFS have approximately the same runtime and memory consumption. The reason for this is that in both cases we perform a full exploration of the state-space. In terms of memory consumption BFS outperforms DFS for the airbag model, the reason for this is the comparably large number of bad states in the airbag model. While BFS finds the shortest counterexamples first, DFS might find longer traces leading to the bad states and, consequently, consumes more memory due to the larger search depth.

- The causality checking adds a runtime and memory penalty, but the experiments show that causality checking is applicable to industrial size Promela

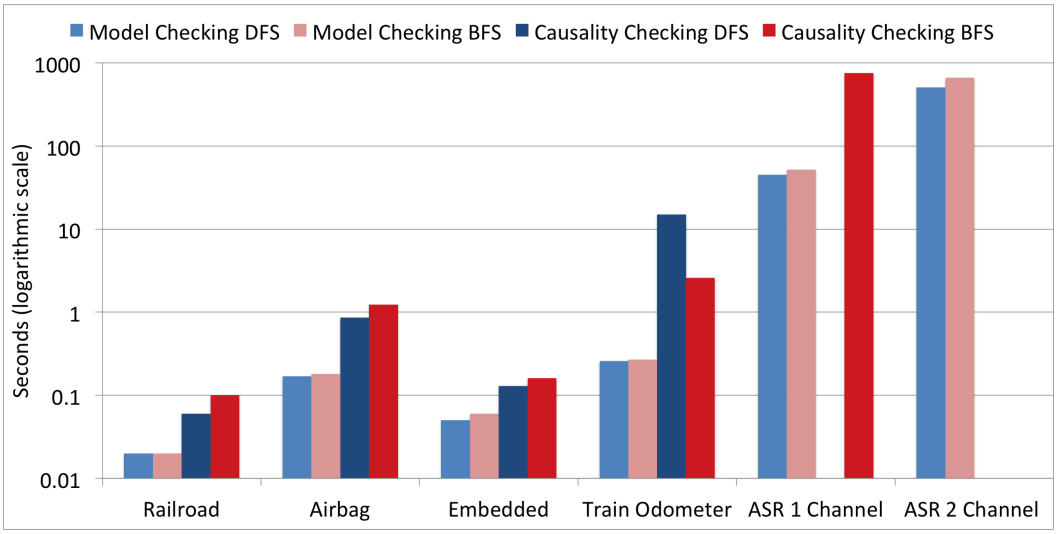


Figure 6.7: Runtime needed for model checking of the case studies with DFS and BFS and for causality checking with DFS and BFS (logarithmic scale).

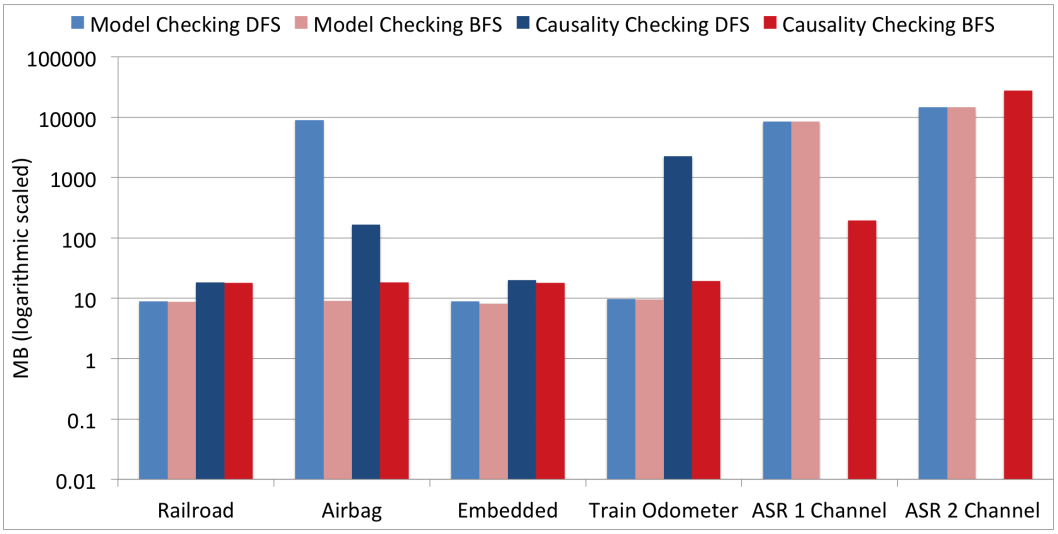


Figure 6.8: Memory needed for model checking of the case studies with DFS and BFS and for causality checking with DFS and BFS (logarithmic scale).

models.

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

- It is not possible to perform causality checking with DFS for both variants of the ASR case study since the algorithm runs out of memory. Causality checking using BFS also runs out of memory for the 2 channel variant but is able to complete for the 1 channel variant of the ASR case study.

	Iterative Approach with Parallel BFS		Iterative Approach with Standard BFS		Standard Approach with BFS	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
Railroad	0.74	17.92	0.83	17.93	0.10	8.97
Airbag	1.59	18.51	1.55	18.53	1.24	21.19
Embedded	0.75	17.99	0.75	17.99	0.16	9.43
Train Odometer	1.44	19.11	1.59	19.21	2.59	63.36
ASR 1 Channel	50.37	195.51	219.88	195.89	750,31	24,663.61
ASR 2 Channel	1,101.99	6,967.00	1,342.69	27,759.32	out of mem.	out of mem.

Table 6.3: Runtime and memory needed for the iterative causality checking approaches and the standard causality checking approach.

In Table 6.3 the runtime and memory needed for the iterative causality checking approach with parallel BFS (using 10 threads), the iterative causality checking approach with standard BFS, and the standard causality checking approach are presented.

The different runtime values are visualized in Figure 6.9 and the memory consumption is visualized in Figure 6.10.

The following trends can be identified:

- For very small models, like the railroad crossing and the embedded control model, both the runtime and the memory consumption for the iterative ap-

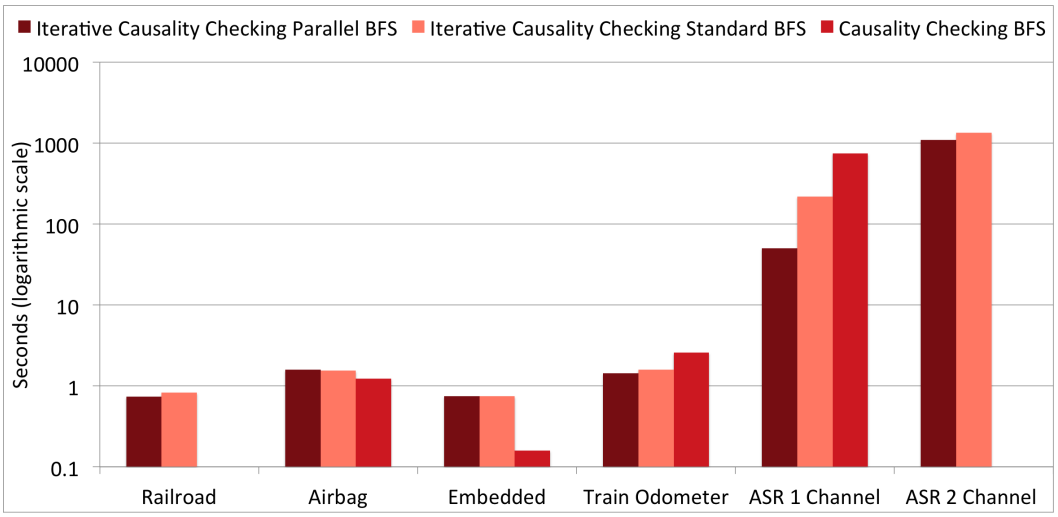


Figure 6.9: Runtime needed for the iterative causality checking approaches and the standard causality checking approach (logarithmic scale).

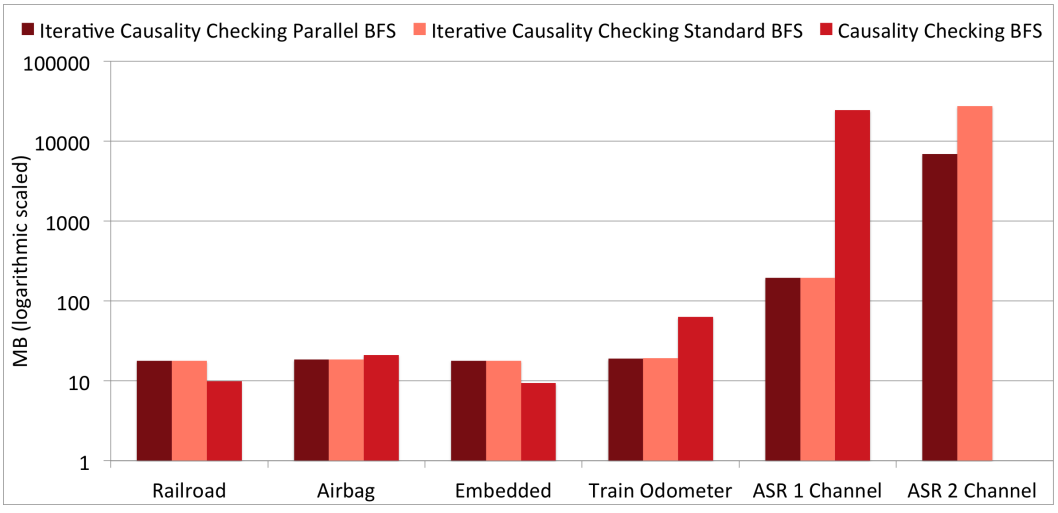


Figure 6.10: Memory needed for the iterative causality checking approaches and the standard causality checking approach (logarithmic scale).

proaches are higher than for the standard approach. This is due to the fact that the state-space is explored twice. With increasing size of the model, the iterative approaches outperform the standard approach with respect to runtime and memory consumption.

- With increasing size of the model the iterative approach with parallel BFS outperforms the iterative approach with the standard BFS in terms of runtime. Note that even though for the parallel BFS approach the different BFS threads need to be managed there is no overhead introduced by the parallelization of BFS.
- While it is not possible to perform a causality checking of the 2 channel variant of the ASR case study with the standard approach, both of the iterative approaches are able to perform causality checking on the 2 channel variant of the ASR case study.

6.6.7 Summary

We have evaluated the usefulness and performance of the proposed causality checking approaches on 5 case studies. Two of the case studies are taken from industrial projects. With the optimized iterative approaches it is possible to perform causality checking for all of the five case studies.

The results generated by the causality checker provide valuable insight as to why the hazard occurred, which is very tedious or even impossible to determine if standard model checking and manual counterexample analysis is used, due to the amount of counterexamples generated.

The experimental evaluation indicates that causality checking using BFS outperforms causality checking using DFS in terms of memory and runtime. For large models, the iterative causality checking approach using parallel BFS outperforms all other causality checking approaches in terms of runtime and memory. Consequently, we recommend to use the iterative causality checking approach using parallel BFS as default algorithm for causality checking of industrial sized models.

Chapter 7

Probabilistic Causality Checking

The content of this chapter is based on the publications [61, 62, 78].

Contents

7.1 Introduction

In the previous chapter we have shown how causality reasoning can be integrated into qualitative model checking. While qualitative causality checking is very helpful for the analysis of systems, it can not give any insights in how much probability mass a certain causality class contributes to the probability of a property violation. As we have argued in Chapter 1, the hardware in which the software is embedded might deteriorate and exhibit certain failure behavior that impacts the software, such as for example bit flips in the memory that is used by the software. In industry negative exponentially distributed rates [96, 95], also called failure rates, are used to estimate the occurrence probability of such hardware failures in a given time frame. In this chapter we extend our causality checking to be applicable to probabilistic models and, consequently, to compute the causal events based on a set of probabilistic counterexamples [5, 47].

7.2 Causality Checking on Probabilistic Counterexamples

We propose a probabilistic causality checking approach that is applied as a post-processing step on a probabilistic counterexample.

The approach presented here is not limited to any probabilistic counterexample generation tool in particular, as long as it is possible to enumerate all traces. The COMICS [56] tool and the DiPro [3] tool are the only publicly available probabilistic counterexample generation tools at the time of this writing. The COMICS toolÃ¶ computes a minimal critical subsystem of the system as counterexample, the critical subsystems are subgraphs of the analyzed Markov chain. The DiPro tool can either enumerate the traces in the counterexample or produce diagnostic subgraphs which are very similar to the notion of the critical subsystems computed by COMICS. Since the approach we present here requires an enumeration of all bad and good traces, we use the DiPro tool in the experimental evaluation section of this chapter. Note that the DiPro tool could be replaced by any other probabilistic counterexample generation tool that allows for the enumeration of all bad and good traces.

It is not possible to implement probabilistic causality checking on-the-fly because all bad traces have to be known in order to compute the probability of the property violation. We use the DiPro tool to compute a complete probabilistic counterexample which contains all paths leading to a state satisfying some property φ. This is achieved by computing the counterexample for the CSL formula $\mathcal{P}_{=?}(\text{true}\,\mathcal{U}\,\varphi)$. Note that in the case of probabilistic causality checking we expect φ to express the undesired system state. The complete probabilistic counterexample together with all good paths which can be obtained at no additional computational effort are taken as an input for the proposed approach.

We interpret counterexamples as a set of execution traces Σ. We assume that Σ contains all execution traces of the model we wish to analyze and that Σ is partitioned in disjoint sets Σ_G and Σ_B. The set Σ_B contains all traces belonging to the counterexample, and the set Σ_G contains all system traces that do not belong to the counterexample. Notice, that in order to compute the full probability of reaching the state where φ holds, it is necessary to perform a complete state-space exploration of the model we analyze. Hence we obtain Σ_G at no additional cost.

Definition 35. *Good and Bad Traces. Let Σ the set of all good and bad traces of the model and φ the hazard.*

- $\Sigma_G = \{\sigma \in \Sigma \mid \sigma \not\models_{CSL} \varphi\}$,
- $\Sigma_B = \{\sigma \in \Sigma \mid \sigma \models_{CSL} \varphi\}$, *and*
- $\Sigma_G \cup \Sigma_B = \Sigma$ *and* $\Sigma_G \cap \Sigma_B = \varnothing$.

Next we define the candidate set of traces that we consider to be causal for φ. The candidate set is defined in such a way that it includes all minimal traces. Traces are minimal if they do not contain a sub-trace according to the $\subseteq$ operator that is also a member of the candidate set.

Definition 36. *Candidate Set. Let Σ_B the set of all bad execution traces with respect to the undesired property φ. We define the candidate set of causal traces φ as CS(φ): $CS(\varphi) = \{\sigma \in \Sigma_B | \forall \sigma' \in \Sigma_B \,.\, \sigma' \subseteq \sigma \Rightarrow \sigma' = \sigma\}$.*

Notice, that the traces in the candidate set are minimal in the sense that removing an event from some trace in the candidate set means that the resulting trace is no longer in the counterexample Σ_B. Each $\sigma \in \Sigma_B$ can be represented by an event order logic formula ψ_σ (c.f. Def. 22).

The pseudocode of the probabilistic causality checking algorithm is shown in Listing 7.1. The computation of the candidate set is performed in lines 11-21 of Listing 7.1 and the isMinimalTrace method shown in lines 42-52 of Listing 7.1.

```
1  INPUT:
2  List of Traces CX; /*bad traces*/
3  List of Traces  G; /*good traces*/
4
5  function ProbabilisticCausalityChecking(CX, G)
6  {
7      List of Traces candidateSet = {};
8      List of Traces notInCandidateSet  = {};
9
10     FOR EACH Trace t in CX
11     {
12         IF(isMinimalTrace(t))
13         {
14             addTo(candidateSet, t);
15         }
16         ELSE
17         {
18             addTo(notInCandidateSet, t);
19         }
20     }
21
22     checkAC22();
23
24     FOR EACH Trace t in candidateSet
25     {
26         checkOC(t);
27     }
28
29     FOR EACH Trace t in notInCandidateSet
30     {
31         if(checkCC(t))
32         {
33             checkOC(t);
34             t.setIsCC(true);
35             addTo(candidateSet, t);
36         }
37     }
38
39     assignProbabilities();
40 }
41
```

```
42 function boolean isMinimalTrace(Trace t)
43 {
44     FOR EACH Trace t' in CX
45     {
46         if(t' is real subset of t)
47         {
48             return false;
49         }
50     }
51     return true;
52 }
```

Listing 7.1: Algorithm sketch of the probabilistic causality checking approach.

We now show that the EOL formula ψ_σ representing a trace $\sigma \in CS(\varphi)$ satisfies the conditions AC1, AC2(1) and AC3.

Theorem 55. *AC1 holds for all ψ_σ with $\sigma \in CS(\varphi)$.*

Proof. According to Definition 36 all traces $\sigma \in CS(\varphi)$ are traces in Σ_B, consequently, $\sigma \vDash_{CSL} \varphi$ holds for all $\sigma \in CS(\varphi)$. And $\sigma \vDash \psi_\sigma$ holds by definition of ψ_σ. Therefore, AC1 holds for all ψ_σ with $\sigma \in CS(\varphi)$. □

In order to show that AC2(1) holds for all ψ_σ with $\sigma \in CS(\varphi)$ we first demonstrate that AC2(1) is fulfilled for a σ if we can find a $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$.

Lemma 1. *AC2(1) holds for ψ_σ if there is a trace $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$.*

Proof. To show AC2(1) for a trace σ we need to show that there exists a trace σ' for which $\sigma' \nvDash \psi \wedge (val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ and $\sigma' \nvDash_{CSL} \varphi$ holds. For each $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$ there is at least one event on σ that does not occur on σ'. Because that missing event is part of ψ_σ and Z it follows $\sigma' \nvDash \psi_\sigma$ and $(val_Z(\sigma) \neq val_Z(\sigma') \vee val_W(\sigma) \neq val_W(\sigma'))$ follows, since the value of the event variable representing the missing event assigned by $val_Z(\sigma)$ is *true* and the value assigned by $val_Z(\sigma')$ is *false*. Therefore, we can show AC2(1) for ψ_σ by finding a trace $\sigma' \in \Sigma_G$ for which $\sigma' \sqsubset \sigma$ holds. □

We are now ready to show that AC2(1) holds for all ψ_σ with $\sigma \in CS(\varphi)$.

Theorem 56. *AC2(1) holds for all ψ_σ with $\sigma \in CS(\varphi)$.*

Proof. According to Lemma 1 AC2(1) holds for ψ_σ if there is a trace $\sigma' \in \Sigma_G$ with $\sigma' \sqsubset \sigma$. Recall that all traces in $CS(\varphi)$ are minimal bad traces, that is all sub-traces of the traces in $CS(\varphi)$ are good traces. Consequently, AC2(1) is fulfilled by all ψ_σ with $\sigma \in CS(\varphi)$. □

We now need to test AC2(2) for σ. AC2(2) requires that $\forall \sigma''$ with $\sigma'' \vDash \psi_\sigma \wedge (val_Z(\sigma) = val_Z(\sigma'') \wedge val_W(\sigma) \neq val_W(\sigma''))$ it holds that $\sigma'' \nvDash_{CSL} \varphi$ for all subsets of W. Suppose there exists a σ'' for which $\sigma \dot{\subset} \sigma''$ holds. For a σ'' to satisfy the

condition $\sigma'' \models \psi \wedge val_Z(\sigma) = val_Z(\sigma'')$ all events that occur on σ have to occur in the same order on σ'', which is the case if $\sigma \dot{\subseteq} \sigma''$ holds. The set W contains the event variables of the events that did not occur on σ and $val_W(\sigma)$ assigns *false* to all event variables in W. For $val_W(\sigma'')$ to be different from $val_W(\sigma)$ there has to be at least one event variable that is set to *true* by $val_W(\sigma'')$. This is only the case if an event that does not occur on σ occurs on σ''. Consequently, σ'' consists of all events that did occur on σ and at least one event that did not occur on σ which is true if $\sigma \dot{<} \sigma''$ holds. $\sigma'' \not\models_{CSL} \varphi$ holds if $\sigma'' \in \Sigma_B$ and is false if $\sigma'' \in \Sigma_G$. Hence, AC2(2) holds for σ if there is no $\sigma'' \in \Sigma_G$ for which $\sigma \dot{<} \sigma''$ holds. If $\sigma \dot{<} \sigma''$ holds for some σ'' we need to modify ψ_σ as specified by Definition 31. The AC2(2) test is performed by the lines 1-50 of Listing 7.2 and called in line 22 of Listing 7.1. For each of the traces in the candidate set, the minimal sets of events that can prevent the property violation on the red trace are computed (Listing 7.2, lines 1-50). This is achieved by comparing the traces in the candidate set with all good traces that are supersets of the trace from the candidate set (Listing 7.2, lines 3-16). For each good superset trace we check whether the set Q of event variables that conjunctively prevent the occurrence of the property violation is minimal and if that is the case add it to the list of Qs (Listing 7.2, lines 31-47).

```
1  function checkAC22()
2  {
3    FOR i = 0 to i < size(candidateSet)
4    {
5      j = 0;
6
7      t' = candidateSet.get(i);
8      Qs = {};
9      t = G.get(j);
10
11     WHILE(j < size(G))
12     {
13         j++;
14         t = G.get(j);
15
16         if((t'.length() < t.length()) && t' is subset of t)
17         {
18             u = 0;
19             Q = {}
20             FOR x = 0 to t.length()
21             {
22                 if(t'.actions.get(u) = t.action.get(x)))
23                 {//events are same move to next event
24                     u++;
25                 }
26                 else if(Q NOT IN t.action.get(x))
27                 {
28                     Q.add(t.action.get(x));
29                 }
```

```
30              }
31              IF(Q != {})
32              {
33                add = true;
34                FOR EACH q in Qs
35                {
36                  IF(Q is subset or equal q)
37                  {
38                       add = false;
39                  }
40                }
41                if(add)
42                {
43                  Qs.add(Q);
44                }
45              }
46          }
47          t'.addQs(Qs);
48      }
49    }
50  }
```

Listing 7.2: Algorithm sketch of the AC2(2) test of the probabilistic causality checking approach.

We will now show that AC3 is fulfilled by all traces in the candidate set $CS(\varphi)$.

Theorem 57. *AC3 holds for all ψ_σ with $\sigma \in CS(\varphi)$.*

Proof. AC(3) requires that no true subset of the event order logic formula ψ satisfies AC1, AC2(1) and AC2(2). Suppose there exists a $\sigma''' \in \Sigma_B$ with $\sigma''' \subset \sigma$. We can partition $\mathcal{A}$ in $Z_{\sigma'''}$ and $W_{\sigma'''}$ such that $Z_{\sigma'''}$ consists of the variables of the events that occur on σ''' and $\psi_{\sigma'''}$ consists of the variables in $Z_{\sigma'''}$. For σ we partition $\mathcal{A}$ in Z_σ and W_σ such that Z_σ consists of the variables of the events that occur on σ and ψ_σ consists of the variables in Z_σ. Consequently, $Z_{\sigma'''} \subset Z_\sigma$ and $\psi_{\sigma'''}$ is a true subset of ψ_σ. If $\psi_{\sigma'''}$ satisfies AC1, AC2(1), AC2(2), then AC3 would be violated. But if such a σ''' would exists, σ would not be in the candidate set, since it would violate the condition that no trace $\sigma''' \in \Sigma_B$ with $\sigma''' \subset \sigma$ exists. Consequently, AC3 is fulfilled for all ψ_σ with $\sigma \in CS(\varphi)$. □

It remains to decide with OC whether the order of the events of the traces $\sigma \in CS(\varphi)$ is relevant to cause φ. For each trace $\sigma \in CS(\varphi)$, we check whether the order of the events to occur is important or not. The implementation of the OC test is shown by the lines 1-24 of Listing 7.3 and is executed for each trace in the candidate set by lines 24-27 of Listing 7.1. For all traces in the candidate set, it is checked whether the candidate set contains another trace containing the same events but representing a different interleaving of the event variables (Listing 7.3, lines 6-8). If such a trace is found, we check for each pair of events whether they appear on

both execution traces in the same order or not (Listing 7.3, lines 10-20). If a pair of events does not occur in the same order, then the order of this pair is marked as having no influence on causality, by adding an entry in the order constraint matrix. After the OC test is completed, the order constraint matrix contains all ordering information and the interleaving can be removed from the candidate set. The EOL formula ψ_σ representing all traces in this causality class is derived from the order constraint matrix.

```
1  function checkOC(Trace t)
2  {
3      //order constraints are iniialized
4      //with true before OC is executed
5
6      FOR EACH Trace t' in candidateSet
7      {
8        if(t.id != t'.id && t is equal t')
9        {
10         FOR i = 0 to size(t.actions)
11         {
12            FOR i = j to size(t.actions)
13            {
14              if(!(t'.actions.indexOf(t.actions.get(i))
15              <= t'.actions.indexOf(t.actions.get(j))))
16              {//pair i,j is not ordered
17                 t.OrderConstraints(i,j,false);
18              }
19            }
20         }
21         remove(candidateSet,t');
22       }
23     }
24 }
```

Listing 7.3: Algorithm sketch of the OC test of the probabilistic causality checking approach.

7.3 Probabilities and Causality Checking

As we have discussed in Section 3.4.3 the probability of the counterexample for a property φ, is the sum of the probabilities of all traces in the counterexample.

Each causality class in the EOL formula Ψ represents a set of traces. Since it is possible that some trace belongs to more than one causality class, it is not possible to compute the probability sum of a causality class by summing up the probabilities of all traces for which the EOL formula representing this causality class is fulfilled. Assume there are two traces $\sigma_1, \sigma_2 \in \Sigma_B$ which, when combined, deliver a trace $\sigma_{1,2} \in \Sigma_B$, and $\sigma_1 \vDash_e \psi_1$, $\sigma_2 \vDash_e \psi_2$, $\sigma_{1,2} \vDash_e \psi_1$, and $\sigma_{1,2} \vDash_e \psi_2$. In this case the

probability of $\sigma_{1,2}$ would be added to the probability of ψ_1 and ψ_2.

To illustrate this point, consider an extension of the railroad example introduced in Section 3.2. We add a traffic light signaling to the car driver that a train is approaching. Event Lr indicates that the traffic light on the crossing is red, while the red light being off is denoted by the event Lo. The traffic light is red whenever a train is approaching and off when the train has left the crossing and no other train is approaching the crossing. It is possible that the traffic light fails (Lf) and in this case remains off although a train is approaching. A car will stop in front of the crossing if the gate is closed and the traffic light is red. Assume that the above described probabilistic causality checking algorithm would identify the following event order logic formulas to be causal: $\mathrm{Gf} \wedge \mathrm{Tc} \wedge \mathrm{Cc}$ and $\mathrm{Lf} \wedge \mathrm{Tc} \wedge \mathrm{Cc}$. The probability of the trace represented by the event order logic formula $\mathrm{Gf} \wedge \mathrm{Lf} \wedge \mathrm{Tc} \wedge \mathrm{Cc}$ would be added to both causality classes, which would lead to a higher over all probability that even could be greater 1.

To account for this situation we introduce the following condition that introduces an exception to the minimality rule of the candidate set defined in Definition 36. The combination condition (CC) will add a causality class to the candidate set even if it is not minimal but it is the combination of two causality classes, the combination condition will not exclude a causality class that already is in the candidate set. The condition is defined as follows.

Definition 37. *Combination Condition (CC). Let σ_1 and σ_2 traces in Σ_B.*

CC: *Let $\sigma_1, \sigma_2, ...\sigma_k \in \mathrm{CS}(\varphi)$ traces and $\psi_{\sigma_1}, \psi_{\sigma_2}, ..., \psi_{\sigma_k}$ the event order logic formulas representing them. A trace σ is added to $\mathrm{CS}(\varphi)$ if for $k \geq 2$ traces in $\mathrm{CS}(\varphi)$ it holds that $(\sigma \vDash \psi_{\sigma_1}) \wedge (\sigma \vDash \psi_{\sigma_2}) \wedge ... \wedge (\sigma \vDash \psi_{\sigma_k})$ and $Z_\sigma = Z_{\sigma_1} \cap Z_{\sigma_2} \cap ... \cap Z_{\sigma_k}$.*

We can now assign each trace in the candidate set the sum of the probability masses of the traces that it represents, since now each bad trace is represented by exactly one causality class in the EOL formula Ψ derived from the candidate set. This is done as follows: The probability of the causality class ψ_i in Ψ, derived from the trace σ_i in *CS(φ)*, is the probability sum of all traces $\sigma' \in \Sigma_B$ that satisfy ψ_i but do not satisfy any other ψ_j in Ψ.

Definition 38. *Probability of a Causality Class Let ψ_i a causality class in Ψ, derived from the trace σ_i in $\mathrm{CS}(\varphi)$, the probability $Pr(\psi_i)$ of ψ_i is defined by*

$$Pr(\psi_i) = \sum_{\sigma \in \Sigma_B \text{ and } \sigma \vDash_e \psi_i \text{ and for } 1 \leq j \leq n \text{ and } j \neq i : \sigma \nvDash_e \psi_j} Pr(\sigma)$$

where n is the number of causality classes in Ψ.

The last condition is necessary in order to correctly assign the probabilities to traces which were added to the candidate set by the CC test. The implementation of the CC test is shown by the lines 1-24 of Listing 7.4 and the assignment of the probabilities is done by the lines 26-56 of Listing 7.4. Both are executed by lines 29-40 of Listing 7.1.

```
1  function boolean checkCC(Trace t)
2  {
3      i = 0;
4      FOR EACH Trace t' in candidateSet
5      {
6          if(t' is real subset of t)
7          {
8              i++;
9          }
10     }
11
12     if(i >= 2)
13     {
14         FOR EACH Trace t'' in notInCandidateSet
15         {
16             if(t.id != t''.id & t'' is subset of t )
17             {
18                 return false;
19             }
20         }
21         return true;
22     }
23     return false;
24 }
25
26 function assignProbabilities()
27 {
28     FOR EACH Trace t in candidateSet
29     {
30         if(t.isCC())
31         {
32             FOR EACH Trace t' in CX
33             {
34                 if(t is subset of t')
35                 {
36                     t.Prob =  t.Prob + t'.Prob;
37                     t'.Prob = 0;
38                 }
39             }
40         }
41     }
42     FOR EACH Trace t in candidateSet
43     {
44         if(!t.isCC())
45         {
46             FOR EACH Trace t' in CX
47             {
48                 if(t is subset of t')
49                 {
```

```
50                          t.Prob =  t.Prob + t'.Prob;
51                          t'.Prob = 0;
52                      }
53                  }
54              }
55          }
56  }
```

Listing 7.4: Algorithm sketch of the CC test and the probability assignment method of the probabilistic causality checking approach.

A rationalization for Definition 37 and Definition 38 based on the definition of the reachability probability from [8] is presented here.

Let $M = (\mathcal{S}, s_0, \mathcal{P}, \mathcal{L})$, a Markov chain where $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{P} : \mathcal{S} \times \mathcal{S} \to \mathbb{P}_{\geq 0}$ is a transition probability matrix and $\mathcal{L} : \mathcal{S} \to 2^{AP}$ is a labeling function, which assigns to each state a subset of the set of atomic propositions AP. The set $B \subseteq \mathcal{S}$ is the set of states satisfying the property φ. We use the probability measure $\Pr^M$ and the cylinder set notion as defined in [8], which we have already introduced in Section 3.4.3,

$$\Pr^M(\mathrm{Cyl}(s_0...s_n)) = P(s_0...s_n)$$

where

$$P(s_0\ s_1...s_n) = \prod_{0 \leq i < n} P(s_i, s_{i+1})$$

and for path fragments of length zero let $P(s_0) = 1$.

Reaching a state in B can be characterized by the union of all basic cylinders $\mathrm{Cyl}(s_0...s_n)$ where $s_0...s_n$ is an initial path fragment in M such that $s_0, ..., s_{n-1} \notin B$ and $s_n \in B$. The set of all such paths is given by $\mathrm{Paths}_{\mathrm{fin}}(M) \cap (\mathcal{S} \setminus B) \times B$. Since these cylinder sets are pairwise disjoint, the probability of eventually reaching a state in B is given by

$$\begin{aligned} \Pr^M(\varphi) &= \sum_{s_0...s_n \in \mathrm{Paths}_{\mathrm{fin}}(M) \cap (\mathcal{S} \setminus B) \times B} \Pr^M(\mathrm{Cyl}(s_0...s_n)) \\ &= \sum_{s_0...s_n \in \mathrm{Paths}_{\mathrm{fin}}(M) \cap (\mathcal{S} \setminus B) \times B} \mathrm{P}(s_0...s_n) \end{aligned}$$

For each of the causality class ψ_i in Ψ derived from the candidate set, we can define a set B_{ψ_i} of states such that for all states in B_{ψ_i} it holds that the traces leading to those states satisfy ψ_i and φ.

Reaching a state in B_{ψ_i} can be characterized by the union of all basic cylinders $\mathrm{Cyl}(s_0...s_n)$ where $s_0...s_n$ is an initial path fragment in M such that $s_0, ..., s_{n-1} \notin B_{\psi_i}$ and $s_n \in B_{\psi_i}$. The set of all such paths is given by $\mathrm{Paths}_{\mathrm{fin}}(M) \cap (\mathcal{S} \setminus B_{\psi_i}) \times B_{\psi_i}$. Since these cylinder sets are pairwise disjoint, the probability of eventually reaching a state in B_{ψ_i} is given by

$$\begin{aligned}\Pr^M(\psi_i) &= \sum_{s_0...s_n \in \text{Paths}_{\text{fin}}(M) \cap (\mathcal{S} \setminus B_{\psi_i}) \times B_{\psi_i}} \Pr^M(\text{Cyl}(s_0...s_n)) \\ &= \sum_{s_0...s_n \in \text{Paths}_{\text{fin}}(M) \cap (\mathcal{S} \setminus B_{\psi_i}) \times B_{\psi_i}} \text{P}(s_0...s_n)\end{aligned}$$

Consequently, the probability $\Pr(\psi_i)$, defined by Definition 38, is equal to $\Pr^M(\psi_i)$.

7.4 Completeness and Soundness

In this section we discuss how the general results for completeness and soundness of causality checking presented in Section 5.3 can be used to show completeness and soundness of the probabilistic causality checking approach.

7.4.1 Completeness

According to Theorem 43 in Section 5.3.1, the event order logic formula returned by the causality checker is complete if all good and bad traces have been identified. The probabilistic causality checking approach is a post-processing step that takes a complete probabilistic counterexample, consisting of all bad traces, and all good traces identified by the probabilistic counterexample generation tool as an input. Consequently, the results generated by the probabilistic causality checking approach are complete.

7.4.2 Soundness

Theorem 44 in Section 5.3.1 makes two assumptions in order to show the soundness of the causality checking result.

1. It is assumed that the bad traces used for the causality checking are sound. The bad traces returned by the probabilistic counterexample generation tool on which we base our causality checking algorithm are sound by definition.

2. It is assumed that all good traces have been found, which is the case for the probabilistic causality checking approach since we require that the probabilistic counterexample generation tool generates all possible good traces as an input for the probabilistic causality checking approach.

It follows that the results generated by the probabilistic causality checking approach are sound.

7.5 Complexity Considerations

The worst-case runtime complexity of the method isMinimalTrace is RT(isMinimalTrace) $\in \mathcal{O}(|t|)$, since it iterates through all traces at most once, and the method is called at most $|t|$ times, where $|t|$ is the total number of traces. The worst-case runtime complexity of the checkAC22 method is RT(checkAC22) $\in \mathcal{O}(|t^2|)$, since the outer and the inner loop iterates through all traces at most once. The checkOC method has a runtime complexity of RT(checkOC) $\in \mathcal{O}(|t|)$, since it iterates through all traces at most once, and the method is called at most $|t|$ times. The checkCC method also has a runtime complexity of RT(checkCC) $\in \mathcal{O}(|t|)$, since it iterates through all traces at most once, and is called at most $|t|$ times. The runtime complexity of the assignProbabilities method is RT(assignProbabilities) $\in \mathcal{O}(|t^2|)$, since the outer and the inner loop iterates through all traces at most once. Consequently, the worst-case runtime complexity of the probabilistic causality checking approach is RT(ProbabilisticCausalityChecking) $\in \mathcal{O}(|t^2|)$.

Similarly to the qualitative causality checking approach, the worst-case in terms of memory consumption is that all traces have to be stored. Consequently, the worst-case memory consumption is in $\mathcal{O}(|t|)$ where $|t|$ is the number of traces.

7.6 Experimental Evaluation

In order to evaluate the proposed probabilistic causality checking approach, we have implemented the approach as a post-processing step of the DiPro [3] tool for probabilistic counterexample generation. The post-processing step computes the causality relationships for a PRISM model and a given CSL property.

We evaluate the probabilistic causality checking approach using the case studies presented in Section 6.6. The qualitative models of the case studies have been extended by negative exponentially distributed rates that model the occurrence probability of the different events in a given time frame. In industry such negative exponentially distributed rates [96, 95], also called failure rates, are used to estimate the occurrence probability of hardware failures in a given time frame. The resulting PRISM models are continuous-time Markov chains. The PRISM models used for the case studies have been created manually, whereas in practical usage scenarios the PRISM models can also be automatically synthesized from higher-level design models, for instance by the QuantUM tool [72]. The following experiments were performed on a PC with two Intel Xeon Processors (4 cores with 3.60 Ghz) and 144 GBs of RAM. For the experiments we set the mission time T of the analyzed properties to 1 hour.

7.6.1 Railroad Crossing

For the railroad crossing the probabilistic causality checking approach generates the same event order logic formula characterizing the causal events for the hazard as the qualitative causality checking approach, namely $\Psi = (\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\barwedge} \mathrm{Cc})) \mathbin{\barwedge}_{<}$

$\neg \mathrm{Cl} \mathbin{\mathbb{A}_>} \mathrm{Tc})) \vee ((\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\mathbb{A}} \mathrm{Cc})) \mathbin{\mathbb{A}_<} \neg \mathrm{Cl} \mathbin{\mathbb{A}_>} (\mathrm{Gc} \wedge \mathrm{Tc}))$. The total probability of the car and the train being on the crossing at the same times is $2.312 \cdot 10^{-4}$, for the causality class $(\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\mathbb{A}} \mathrm{Cc})) \mathbin{\mathbb{A}_<} \neg \mathrm{Cl} \mathbin{\mathbb{A}_>} \mathrm{Tc}))$ the probability is $3.464 \cdot 10^{-5}$ and for the causality class $(\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\mathbb{A}} \mathrm{Cc})) \mathbin{\mathbb{A}_<} \neg \mathrm{Cl} \mathbin{\mathbb{A}_>} (\mathrm{Gc} \wedge \mathrm{Tc}))$ the probability is $1.960 \cdot 10^{-4}$.

7.6.2 Airbag System

The event order logic formula returned by the probabilistic causality checking approach for the inadvertent deployment of the airbag is Ψ = (FASICShortage) $\vee$ (FETStuckHigh $\mathbb{A}$ FASICStuckHigh) $\vee$ (MicroControllerFailure $\mathbb{A}$ enableFET $\mathbb{A}$ armFASIC $\mathbb{A}$ fireFASIC) $\vee$ (FETStuckHigh $\mathbb{A}$ MicroControllerFailure $\mathbb{A}$ armFASIC $\mathbb{A}$ fireFASIC)$\vee$(MicroControllerFailure$\mathbb{A}$enableFET$\mathbb{A}$FASICStuckHigh) which is equal to the event order logic formula returned by the qualitative causality checking. Note that the used probability values are mock numbers, since the real values are intellectual property of our industrial partner. The total probability of an inadvertent deployment is $1.8009 \cdot 10^{-3}$ and is caused by the following causality classes

- (FASICShortage) with a probability of $2.5614 \cdot 10^{-5}$,
- (FETStuckHigh $\mathbb{A}$ FASICStuckHigh) with a probability of $1.6924 \cdot 10^{-6}$,
- (MicroControllerFailure $\mathbb{A}$ enableFET $\mathbb{A}$ armFASIC $\mathbb{A}$ fireFASIC) with a probability of $1.7705 \cdot 10^{-3}$,
- (FETStuckHigh $\mathbb{A}$ MicroControllerFailure $\mathbb{A}$ armFASIC $\mathbb{A}$ fireFASIC) with a probability of $1.3272 \cdot 10^{-6}$, and
- (MicroControllerFailure $\mathbb{A}$ enableFET $\mathbb{A}$ FASICStuckHigh) with a probability of $1.7427 \cdot 10^{-6}$.

7.6.3 Embedded Control System

The event order logic formula returned by the probabilistic causality checker for a shutdown of the embedded control system is Ψ = MainProcFail $\vee$ InputProcFail $\vee$ OutputProcFail $\vee$ InputProcTransFail $\vee$ (SensorFailure $\wedge$ SensorFailure) $\vee$ (ActuatorFailure $\wedge$ ActuatorFailure) which is equal to the event order logic formula returned by the qualitative causality checking.

The total probability is $3.0682 \cdot 10^{-4}$ and can be broken down into

- (MainProcFail) with a probability of $1.1415 \cdot 10^{-4}$,
- (InputProcFail) with a probability of $9.3209 \cdot 10^{-5}$,
- (OutputProcFail) with a probability of $9.3205 \cdot 10^{-5}$,
- (InputProcTransFail) with a probability of $1.0083 \cdot 10^{-11}$,

- (SensorFailure $\wedge$ SensorFailure) with a probability of $5.7759 \cdot 10^{-6}$, and
- (ActuatorFailure $\wedge$ ActuatorFailure) with a probability of $4.8185 \cdot 10^{-7}$.

7.6.4 Train Odometer Controller

The event order logic formula returned by the probabilistic causality checker is

$$\begin{aligned}\Psi =&(\text{Start\_W\_Fail\_S} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_S}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail})\\ &\vee\\ &(\text{Start\_W\_Fail\_F} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_F}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail})\\ &\vee\\ &\text{Wait\_R\_Fail} \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}\\ &\vee\\ &(\text{Wait\_R\_Fail} \wedge (\text{Start\_W\_Fail\_S} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_S})\\ &\wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))\\ &\vee\\ &(\text{Wait\_R\_Fail} \wedge (\text{Start\_W\_Fail\_F} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_F})\\ &\wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))\end{aligned}$$

The two causality classes $(\text{Wait\_R\_Fail} \wedge (\text{Start\_W\_Fail\_S} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_S}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))$ and $(\text{Wait\_R\_Fail} \wedge (\text{Start\_W\_Fail\_F} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_F}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))$ are added by the CC condition in order to correctly assign all probabilities. The total probability is $2.8232 \cdot 10^{-2}$ which can be broken down into

- $((\text{Start\_W\_Fail\_S} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_S}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))$ with a probability of $8.0313 \cdot 10^{-3}$,
- $((\text{Start\_W\_Fail\_F} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_F}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))$ with a probability of $5.5040 \cdot 10^{-4}$,
- $(\text{Wait\_R\_Fail} \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail})$ with a probability of $9.5438 \cdot 10^{-3}$,
- $(\text{Wait\_R\_Fail} \wedge (\text{Start\_W\_Fail\_S} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_S}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))$ with a probability of $7.5087 \cdot 10^{-3}$, and
- $(\text{Wait\_R\_Fail} \wedge (\text{Start\_W\_Fail\_F} \mathbin{\hat{\wedge}} \text{Wait\_W\_Fail\_F}) \wedge (\neg\text{failureDeteted} \mathbin{\hat{\wedge}_]} \text{Wait\_Mon\_Fail}))$ with a probability of $2.5978 \cdot 10^{-3}$.

7.6.5 Airport Surveillance Radar

The probabilistic causality checking approach is not able to compute any results for the airport surveillance radar, since the probabilistic counterexample generation with DiPro runs out of memory.

7.6.6 Discussion

The probabilities computed for the different causality classes of the case studies allow to identify the event combinations that contribute the highest probability to the violation of the system and thus provide valuable insight.

	Counterexample Computation		Causality Computation		Runtime Sums Memory Maximums	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
Railroad	1.00	10.00	1.00	10.00	2.00	10.00
Airbag	606.00	389.00	3.00	233.00	609.00	389.00
Embedded	1,933.00	383.00	3.00	235	1,936.00	383.00
Train Odometer	17,512.00	1,739.00	2.00	239	17,154.00	1,739.00
ASR 1 Channel	out of mem.	out of mem.	N/A	N/A	N/A	N/A
ASR 2 Channel	out of mem.	out of mem.	N/A	N/A	N/A	N/A

Table 7.1: Runtime and memory consumption of the probabilistic counterexample generation and causality checking.

Table 7.1 shows the runtime and memory required for the probabilistic counterexample generation with DiPro and the runtime and memory consumption for the subsequent probabilistic causality checking approach. The runtime and memory consumption for the ASR case study with 1 and 2 channels can not be given, since DiPro ran out of memory (out of mem.) while computing the probabilistic counterexample and, consequently, the subsequent causality checking could not be executed. The relatively short runtime for the causality computation for the train odometer model is caused by the relatively small number of causality classes.

The different runtime values are visualized in Figure 7.1 and the memory consumption is visualized in Figure 7.2.

The experiments show that the computational effort is dominated by the counterexample computation. The main reason for the comparably high amount of runtime and memory needed for the probabilistic counterexample generation lies in the fact that DiPro needs to compute the probability of each bad trace. This means that for each bad trace that is identified a probabilistic model checking run has to be executed in order to compute the probability of the trace. This shortcoming is independent from the DiPro tool, since all probabilistic counterexample generation tools that enumerate all traces have to perform a probabilistic model checking run for each bad trace and the runtime and memory consumed by the underlying probabilistic model checker is the same regardless of which probabilistic counterexample generation tool is used.

7.6.7 Summary

We have evaluated the usefulness and performance of the proposed probabilistic causality checking approaches on 5 case studies. Two of the case studies are taken from industrial projects. While the evaluation shows that the probabilities for the

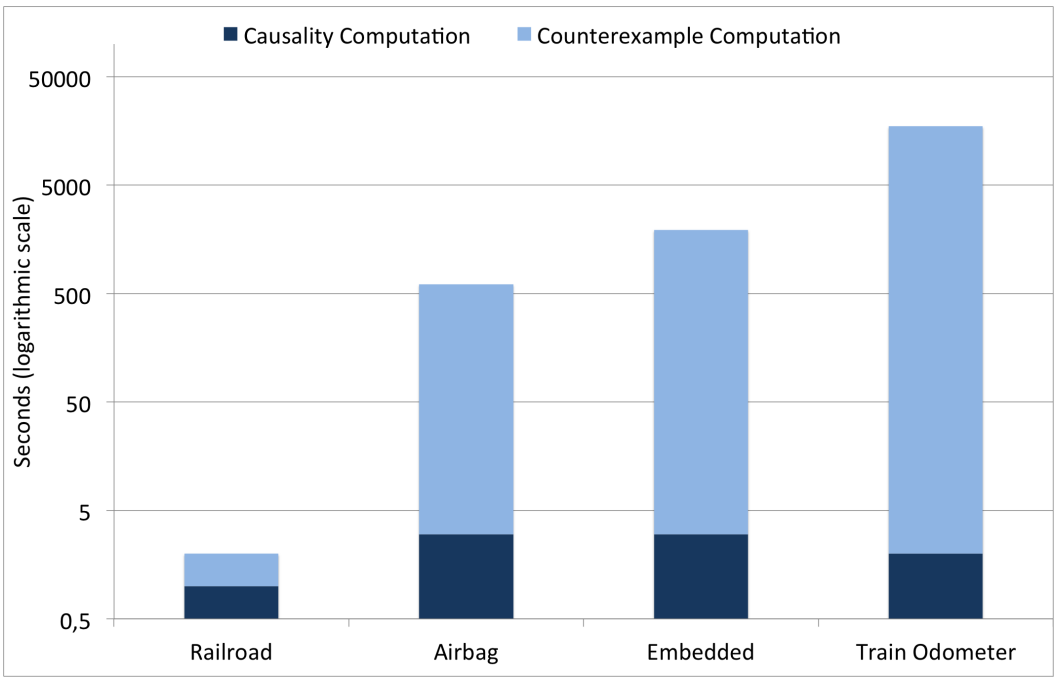


Figure 7.1: Runtime needed for the probabilistic counterexample generation and causality checking (logarithmic scale).

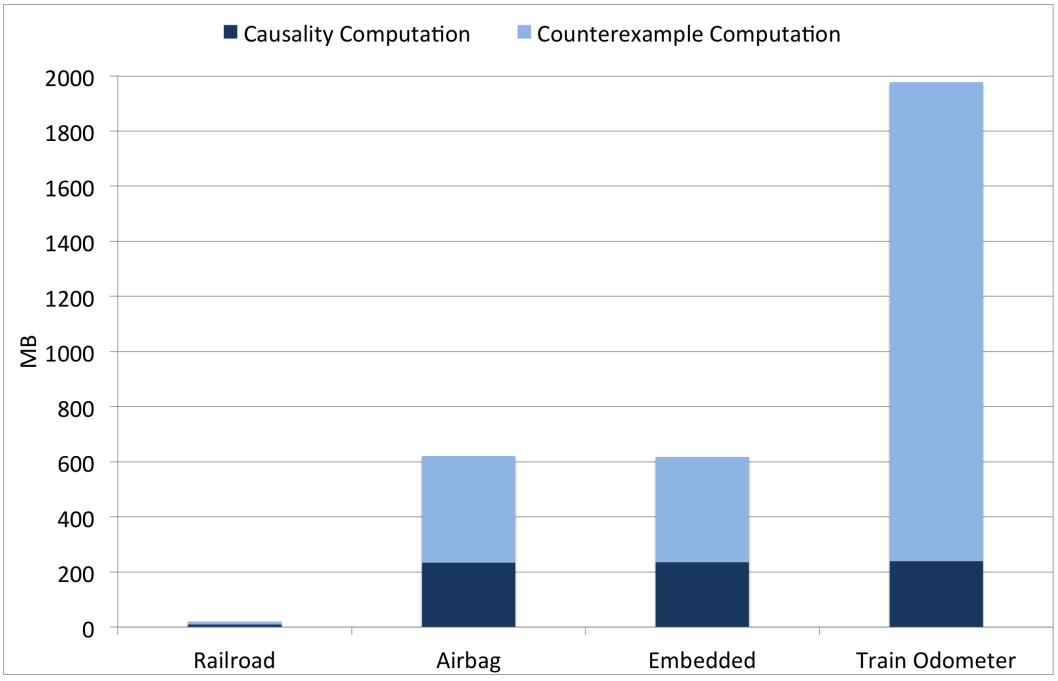


Figure 7.2: Memory consumption of the probabilistic counterexample generation and causality checking.

different causality classes provide an important additional insight as compared to qualitative causality checking, the evaluation also shows that the necessary probability computation for each bad traces introduces a very high runtime and memory penalty. Due to this runtime and memory penalty the probabilistic causality checking approach was not able to compute results for the airport surveillance radar case study.

Chapter 8

Combined Qualitative and Probabilistic Causality Checking

The content of this chapter is based on the publications [77, 79].

Contents

8.1 Introduction

In the previous two chapters we have developed two approaches for causality checking:

1. The *qualitative causality checking* approach described in Chapter 6, where the causality computation algorithm is integrated into the state-space exploration algorithms used for qualitative model checking. This algorithm is capable of computing the causality relationships on-the-fly.

2. The *probabilistic causality checking* approach described in Chapter 7, where causal relationships of events are algorithmically inferred from probabilistic counterexamples.

The main advantage of the probabilistic causality checking approach over the qualitative causality checking approach is that it computes a quantitative measure, namely a probability, for a combination of causal events and hazards to occur. The probability of an event combination causing a property violation to occur is an information that is needed for the reliability and safety analysis of safety-critical systems. A shortcoming of the probabilistic causality checking approach compared

to the qualitative causality checking approach however is, that the causality computation requires a complete probabilistic counterexample consisting of all bad traces and all good traces. The high amount of runtime and memory needed to compute the probabilities of all traces in the probabilistic counterexample limits the scalability of the probabilistic causality checking approach.

The goal of this chapter is to leverage the qualitative causality checking approach in order to improve the scalability of the probabilistic causality checking approach. The key idea is to first compute the causal events using the qualitative causality checking approach and to then limit the probability computation to the causal event combinations that have first been computed. Our proposed combined approach can be summarized by identifying the following steps:

1. The probabilistic PRISM model is mapped to a qualitative Promela model.

2. The qualitative causality checking approach is applied to the qualitative model in order to compute the event combinations that are causal for the property violation.

3. The information obtained through qualitative causality checking is mapped back via alternating automata to the probabilistic model. The probabilities for the different event combinations that are causal for the property violation to occur are computed using a probabilistic model checker.

We discuss the translation of probabilistic PRISM models to qualitative Promela models in Section 8.2. Section 8.3 discusses the translation of the information returned by the causality checker to the PRISM model and the probability computation of the causal events. The soundness and completeness of the combined approach is discussed in Section 8.4 and complexity considerations are presented in Section 8.5. In Section 8.6 we evaluate the usefulness of the proposed approach on several case studies.

8.2 Translating PRISM Models to Promela Models

Our goal is to compute the causal events using the qualitative causality checking approach and limit the probability computation to the causal events. To achieve this goal we need to translate the model given by a continuous-time Markov chain (CTMC) specified in the PRISM language to a labeled transition system in the Promela language. The translation is necessary since the causality checking approach is based on the SpinJa toolset.

Furthermore, the reachability property specified in continuous stochastic logic (CSL) needs to be translated into a formula in linear temporal logic (LTL). The translation of the CSL formula to an LTL formula is straight forward: If the CSL formula is a state formula, then it is also an LTL formula. If the CSL formula is

a path formula, then the path formula is an LTL formula if we replace a bounded-until operator included in the formula with an LTL until operator. CSL formulas containing nested path-operators are outside the scope of this work.

We base our translation of PRISM models to Promela models on the work in [90], but since no implementation of the described approach is available and the approach translates Markov Decision Processes specified in a PRISM model to a Promela model, we can not apply this approach directly. Furthermore, the translation of synchronizing action labels to rendezvous channel chaining in Promela proposed in [90] is not consistent with the PRISM semantics specified in [49]. Our translation algorithm maps the CTMC to a transition system. This mapping is achieved through the transition system that is induced by the CTMC.

Definition 39. *Transition System Induced by a CTMC. Let* $C = (\mathcal{S}, s_0, \mathcal{R}, \mathcal{L})$ *a CTMC. Then* $T = (S, Act, \rightarrow, I, AP, L)$ *is the transition system induced by* C *if:*

- *The set* S *of states in* T *is* $S = \mathcal{S}$.
- *The set* I *of initial states in* T *is* $I = \{s_0\}$.
- *For all pairs* $s, s' \in S$ *we add a transition to* $\rightarrow$ *and a corresponding action to* Act *if* $\mathcal{R}(s, s') > 0$.

We translate the transition system induced by the CTMC specified in the PRISM language into the Promela language. The implementation of the PRISM to Promela translation works on the syntactic level of the PRISM code. PRISM *modules* are translated to *active proctypes* in Promela consisting of a *do*-block which contains the transitions. Transitions that are synchronized are translated according to the parallel composition semantics of PRISM [49]. All variables in the PRISM model are translated to global variables of the corresponding type in the Promela model. This is necessary, since otherwise it would not be possible to read variables from other proctypes as it is permitted in PRISM. Listing 8.1 shows the output of the PRISM to Promela translation of the PRISM code in Listing 3.2 from Section 3.4.4. The comments at the end of each transition are merely added to make the Promela model more readable but are not necessary for the translation.

Our approach requires that each command in the PRISM module is labeled with an action label representing the occurrence of an event. If a command of the PRISM model is not already labeled with an action label a unique action label is added to this command during the translation. This does not change the behavior of the PRISM model since the action label is unique and, consequently, is not synchronized with any other command.

Listing 8.2 shows the PRISM code of the car module of the railroad crossing example. The Promela model generated by the PRISM to Promela translation is shown in Listing 8.3. The variables of the PRISM module (Listing 8.2, lines 2-3) are translated to global variables in the Promela model (Listing 8.3, lines 1-2). The transition statements of the PRISM model (Listing 8.2, lines 4-9) are translated into a do-loop in the Promela model (Listing 8.3, lines 5-15).

Now that we can translate the PRISM model to a Promela model we can apply the qualitative causality checking approach. How the results of the qualitative causality checking can be mapped back to the PRISM model and used for probability computation is discussed in Section 8.3.

```
1  bool var1 = false; byte var2 = 0; byte var3 = 0;
2  active proctype moduleA(){
3    do
4    :: atomic {((var3<2) && (var2<4))
5                -> var2=var2+1; var3=var3+1;}/*Count*/
6    :: atomic {((var3==2) && (var2<4))
7                -> var2=var2+1; var3=0;}/*Count*/
8    :: atomic {(var2==4)
9                -> var1=true;}/*End*/
10   od;}
11 active proctype moduleB(){
12   do
13   :: atomic {((var2<4) && (var3<2))
14               -> var3=var3+1; var2=var2+1;}/*Count*/
15   :: atomic {((var2<4) && (var3==2))
16               -> var3=0; var2=var2+1;}/*Count*/
17   od;
18 }
```

Listing 8.1: Example Promela translation of the PRISM model from Section 3.4.4.

```
1  module car
2   s_car : [0..2] init 0;
3   car_crossing : bool init false;
4   [Ca] s_car = 0
5       -> 0.01 : (s_car' = 1);
6   [Cc] s_car = 1 & gate_open
7       -> 0.1 : (s_car' = 2)&(car_crossing' = true);
8   [Cl] s_car = 2
9       -> 0.1 : (s_car' = 0) & (car_crossing' = false);
10 endmodule
```

Listing 8.2: PRISM model of the railroad example.

```
byte s_car = 0;
bool car_crossing = false;

active proctype car(){
  do
  :: atomic {(s_car==0)
    -> s_car=1; /*car_aproaching*/
    }
  :: atomic {(s_car==1&&gate_open)
    -> s_car=2; car_crossing=true; /*car_crossing*/
    }
  :: atomic {(s_car==2)
    -> s_car=0; car_crossing=false; /*car_troughgate*/
    }
  od;
}
```

Listing 8.3: Promela model of the railroad example.

8.3 Probability Computation for Causality Classes

Each causality class returned by the causality checker represents an equivalence class of bad traces. We can leverage this fact and compute the probability sum of all traces represented by a causality class instead of computing the probability of all traces belonging to this class individually. This means that the number of probabilistic model checking runs is reduced to the number of causality classes instead of the number of traces in the counterexample.

We will now show how the probability sum of all traces represented by a causality class can be computed using the PRISM model checker. In order to compute the probability of all traces represented by a causality class we leverage the fact that each EOL formula can be translated into an alternating automaton as we have shown in Section 4.4. Thus it is possible to construct an alternating automaton for the EOL formula representing a causality class such that the alternating automaton accepts exactly those execution traces that are represented by the corresponding causality class.

For the computation of the probability of a causality class we need to translate the corresponding alternating automaton into the PRISM language and synchronize it with the PRISM model.

Note that we only consider non-reachability properties in this thesis. A non-reachability property is violated as soon as a bad state is reached, and no future event can prevent the violation. Consequently, it can not be the case that an event voiding causality, by violating the AC2(2) condition, appears at the end of an execution trace. The EOL operator $\wedge_{[}$ can hence not be added to an EOL formula as a consequence of AC2(2). Therefore, all alternating automata generated from an EOL formula are alternating automata on finite words since they represent finite

execution fragments. Notice that the only way for a $\neg$ operator to be added to an EOL formula by the causality checking algorithm is when the non-occurrence of the an event in the specified interval is causal.

Each action label in the PRISM model corresponds to an event variable in the set $\mathcal{A}$ over which the EOL formulas were built. As a consequence each alternating automaton accepts a finite sequence of PRISM action labels.

We now define translation rules from alternating automata to PRISM modules. We call a PRISM module that was generated from an alternating automaton a *causality class module*. The transitions of the causality class modules are synchronized with the corresponding transitions of the PRISM model. The transition rates of the causality class modules are set to 1.0, as a consequence, the transitions synchronizing with the causality class modules define the rate for the synchronized transition. In Listing 8.4 we present the pseudocode of the algorithm that generates a causality class module from an alternating automaton representing a causality class. The order constraints specified by the EOL formula are encoded by guards. Since we use guards to enforce the order constraint no distinction between alternating automata generated from an EOL formula containing an ordered operator ($A_{\vartriangleleft}(\psi)$) and alternating automaton generated from an EOL formula not containing an ordered operator ($A(\psi)$) has to be made.

In the line 1 of Listing 8.4 the variables that store the output are initialized, in line 4 the PRISM_CODE method is called in order to translated the alternating automaton, and in lines 5-6 the PRISM module representing the alternating automaton is written. The PRISM_CODE method selects the translation rules for a specific alternating automaton and recursively calls itself until the full alternating automaton is translated (Listing 8.4, lines 8-69).

Synchronized transitions can only be executed, if for each other module containing transitions with the same action label the guard of at least one transition per module evaluates to true. It might hence be the case that the causality class module prevents the execution of transitions in the PRISM model with which the causality class module is synchronized. Since this would change the behavior of the PRISM model and affect the probability mass distribution we add a transition with the negated guard and without updates for each transition of the causality class module for which the guard is not always true.

```
1  global var var_def = "", trans = "", formulas = "";
2  function EOL_TO_PRISM(A(ψ))
3  {
4    PRISM_CODE(A(ψ),true)
5    print "module ψ \n" + var_def +"\n"+ trans
6          + " \n endmodule \n" + formulas;
7  }
8  function PRISM_CODE(A(ψ), cond)
9  {
10   IF A(ψ) = 'A(a)' THEN
11      var_def += 's_ψ: bool init false;'
12      IF cond = 'true' THEN
```

```
13        trans += '[a] (cond) -> 1.0 : (s_ψ'=true);'
14      ELSE
15        trans += '[a] (cond) -> 1.0 : (s_ψ'=true);'
16        trans += '[a] !(cond) -> 1.0 : true;'
17      ENDIF
18      formulas += 'formula acc_ψ = s_ψ;'
19    ELSE IF A(ψ) = 'A(¬a)' THEN
20      var_def += 's_ψ: bool init true;'
21      IF cond = 'true' THEN
22        trans += '[a] (cond) -> 1.0 : (s_ψ'=false);'
23      ELSE
24        trans += '[a] (cond) -> 1.0 : (s_ψ'=false);'
25        trans += '[a] !(cond) -> 1.0 : true;'
26      ENDIF
27      formulas += 'formula acc_ψ = s_ψ;'
28    ELSE IF A(ψ) = 'A(ϕ1 ∧ ϕ2)' THEN
29      PRISM_CODE(A(ϕ1), cond);
30      PRISM_CODE(A(ϕ2), cond);
31      formulas += 'formula acc_ψ = acc_ϕ1 & acc_ϕ2;'
32    ELSE IF A(ψ) = 'A(¬(ϕ1 ∧ ϕ2))' THEN
33      PRISM_CODE(A(¬ϕ1), cond);
34      PRISM_CODE(A(¬ϕ2), cond);
35      formulas += 'formula acc_ψ = acc_¬ϕ1 & acc_¬ϕ2;'
36    ELSE IF A(ψ) = 'A(ϕ1) ∧ A(ϕ2)' THEN
37      PRISM_CODE(A(ϕ1), cond);
38      PRISM_CODE(A(ϕ2), cond);
39      formulas += 'formula acc_ψ = acc_ϕ1 & acc_ϕ2;'
40    ELSE IF A(ψ) = 'A(ψ∧1 ∧ ψ∧2)' THEN
41      PRISM_CODE(A(ψ∧1), cond);
42      PRISM_CODE(A(ψ∧2), cond);
43      formulas += 'formula acc_ψ = acc_ψ∧1 & acc_ψ∧2;'
44    ELSE IF A(ψ) = 'A(ψ∧1) ∧ A(ψ∧2)' THEN
45      PRISM_CODE(A(ψ∧1), cond);
46      PRISM_CODE(A(ψ∧2), cond);
47      formulas += 'formula acc_ψ = acc_ψ∧1 & acc_ψ∧2;'
48    ELSE IF A(ψ) = 'A(ψ∧1 ∨ ψ∧2)' THEN
49      PRISM_CODE(A(ψ∧1), cond);
50      PRISM_CODE(A(ψ2), cond)
51      formulas += 'formula acc_ψ = acc_ψ∧1 | acc_ψ2;'
52    ELSE IF A(ψ) = 'A(ψ∧1) ∨ A(ψ2)' THEN
53      PRISM_CODE(A(ψ∧1), cond);
54      PRISM_CODE(A(ψ∧2), cond);
55      formulas += 'formula acc_ψ = acc_ψ1 | acc_ψ∧2;'
56    ELSE IF A(ψ) = 'A(ϕ1 ∧ ϕ2)' THEN
57      PRISM_CODE(A(ϕ1), cond);
58      PRISM_CODE(A(ϕ2), acc_ϕ1);
59      formulas += formula acc_ψ = acc_ϕ2;
60    ELSE IF A(ψ) = 'A(ϕ1 ∧] ϕ2)' THEN
61      PRISM_CODE(A(¬ϕ1), cond);
```

```
62      PRISM_CODE(A(φ2), cond & !(acc_¬φ1));
63      formulas += 'formula acc_ψ = acc_φ2;'
64    ELSE IF A(ψ) = 'A(φ1 ∧< φ∧ ∧> φ2)' THEN
65      PRISM_CODE(A(φ1), cond);
66      PRISM_CODE(A(¬φ∧), acc_φ1)
67      PRISM_CODE(A(φ2), (acc_φ1 & !(acc_¬φ∧))
68      formulas += 'formula acc_ψ = acc_φ2;'
69    ENDIF
70  }
```

Listing 8.4: Pseudocode of the EOL to PRISM algorithm.

We also add a PRISM *formula* acc_ψ for each sub-automaton which is true whenever the corresponding sub-automaton is accepting the input word. Those formulas are used to construct a CSL formula of the form $\mathcal{P}_{=?}[(true)U(\text{acc\_}\psi)]$ for each causality class. The CSL formulas can then be used to compute the probability of all possible traces that are accepted by the causality class module, which is exactly the probability sum of all traces that are represented by the causality class.

Theorem 58. *The probability computed by* $\mathcal{P}_{=?}[(true)U(acc\_\psi)]$ *is equal to the probability sum of all traces* σ *for which* $\sigma \vDash_e \psi$ *holds.*

Proof. Let $M = (\mathcal{S}, s_0, \mathcal{P}, \mathcal{L})$, a Markov chain where $\mathcal{S}$ is a finite set of states, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{P} : \mathcal{S} \times \mathcal{S} \to \mathbb{P}_{\geq 0}$ is a transition probability matrix and $\mathcal{L} : \mathcal{S} \to 2^{AP}$ is a labeling function, which assigns to each state a subset of the set of atomic propositions AP. The set $B \subseteq \mathcal{S}$ is the set of states satisfying the property acc_ψ. We use the probability measure $\Pr^M$ and the cylinder set notion as defined in [8], which we have already introduced in Section 3.4.3,

$$\Pr^M(\text{Cyl}(s_0...s_n)) = P(s_0...s_n)$$

where

$$P(s_0\ s_1...s_n) = \prod_{0 \leq i < n} P(s_i, s_{i+1})$$

and for path fragments of length zero let $P(s_0) = 1$.

Reaching a state in B can be characterized by the union of all basic cylinders $\text{Cyl}(s_0...s_n)$ where $s_0...s_n$ is an initial path fragment in M such that $s_0, ..., s_{n-1} \notin B$ and $s_n \in B$. The set of all such paths is given by $\text{Paths}_{\text{fin}}(M) \cap (\mathcal{S} \setminus B) \times B$. Since these cylinder sets are pairwise disjoint, the probability of eventually reaching a state in B is given by

$$\begin{aligned}\Pr^M((true)U(\text{acc\_}\psi)) &= \sum_{s_0...s_n \in \text{Paths}_{\text{fin}}(M) \cap (\mathcal{S} \setminus B) \times B} \Pr^M(\text{Cyl}(s_0...s_n)) \\ &= \sum_{s_0...s_n \in \text{Paths}_{\text{fin}}(M) \cap (\mathcal{S} \setminus B) \times B} \text{P}(s_0...s_n)\end{aligned}$$

Consequently, the $\Pr^M((true)U(\text{acc\_}\psi))$ is the probability sum of all paths leading to a state in B that accepts acc_ψ and this is only the case if the path is an

accepting run of the PRISM causality class module, since otherwise acc_ψ would not be true in this state. □

Since it is possible that a trace belongs to more than one causality class, we add an additional CSL formula that computes the probability of all traces that are only in the causality class defined by ψ. This CSL formula has the form of $\mathcal{P}_{=?}[(true)U(\text{acc\_}\psi)\&!(\text{acc\_}\psi_i|...|\text{acc\_}\psi_j))]$, where acc_$\psi_i$|...|acc_$\psi_j$ are the formulas of all causality classes except ψ.

Listing 8.5 shows the PRISM code of the EOL formula $(\text{Ta} \wedge (\text{Ca} \barwedge \text{Cc})) \barwedge_{<} \neg\text{Cl} \barwedge_{>} (\text{Gc} \wedge \text{Tc})$ of the railroad crossing example. In lines 2-7 of Listing 8.5 the variables that are needed to store the different events are declared and initialized. The transitions in lines 8-17 of Listing 8.5 are synchronized with the rest of the model and detect the occurrence of events. The guards used in these transitions are the formulas defined in lines 19-26 of Listing 8.5 that indicate in which state the PRISM causality class module is. The formula acc_train_cc_2 in line 27 of Listing 8.5 is true whenever the PRISM causality class module is in an accepting state.

```
1  module train_cc_2
2    s_Ta : bool init false;
3    s_Ca : bool init false;
4    s_Cc : bool init false;
5    s_Cl : bool init true;
6    s_Tc : bool init false;
7    s_Gc : bool init false;
8    [Ta] (true) -> 1.0 : (s_Ta'=true);
9    [Ca] (true) -> 1.0 : (s_Ca'=true);
10   [Cc] (acc_Ca) -> 1.0 : (s_Cc'=true);
11   [Cc] !(acc_Ca) -> 1.0 : true;
12   [Cl] (acc_Ta_Ca_Cc) -> 1.0 : (s_Cl'=false);
13   [Cl] !(acc_Ta_Ca_Cc) -> 1.0 : true;
14   [Gc] (acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : (s_Gc'=true);
15   [Gc] !(acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : true;
16   [Tc] (acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : (s_Tc'=true);
17   [Tc] !(acc_Ta_Ca_Cc & !acc_Cl) -> 1.0 : true;
18 endmodule
19 formula acc_Ta = s_Ta;
20 formula acc_Ca = s_Ca;
21 formula acc_Ca_Cc = s_Ca & s_Cc;
22 formula acc_Ta_Ca_Cc = acc_Ta & acc_Ca_Cc;
23 formula acc_Cl = s_Cl;
24 formula acc_Gc = s_Gc;
25 formula acc_Tc = s_Tc;
26 formula acc_Gc_Tc = acc_Gc & acc_Tc;
27 formula acc_train_cc_2 = acc_Gc_Tc;
```

Listing 8.5: PRISM code of the EOL formula $(\text{Ta} \wedge (\text{Ca} \barwedge \text{Cc})) \barwedge_{<} \neg\text{Cl} \barwedge_{>} (\text{Gc} \wedge \text{Tc})$.

In the railroad example, the total probability of a state where both the train

and the car are on the crossing is p_total = $2.312 \cdot 10^{-4}$. The proposed combined approach returns for the causality class characterized by ψ_1 = Gf $\wedge$ ((Ta $\wedge$ (Ca $\wedge$ Cc)) $\wedge_<$ $\neg$Cl $\wedge_>$ Tc) the total probability of p_{ψ_1} = $3.464 \cdot 10^{-5}$ and the exclusive probability of p_{ψ_1}_excl = $3.464 \cdot 10^{-5}$, and for the causality class characterized by ψ_2 = (Ta $\wedge$ (Ca $\wedge$ Cc)) $\wedge_<$ $\neg$Cl $\wedge_>$ (Gc $\wedge$ Tc) the total probability of p_{ψ_2} = $1.960 \cdot 10^{-4}$ and the exclusive probability of p_{ψ_2}_excl = $1.960 \cdot 10^{-4}$.

8.4 Completeness and Soundness

The proposed combined approach uses the qualitative causality checking approach described in Chapter 6 in order to identify the causal event combinations. We have shown in Section 5.3.1 that the qualitative causality checking approach is sound and complete if a full exploration of the model is possible and all bad and good traces can be identified. Consequently, the combined approach is sound and complete if during the causality checking step a full exploration of the model is possible and all bad and good traces can be identified.

8.5 Complexity Considerations

The worst-case runtime of the combined approach is the combined worst-case runtime of the qualitative causality checking approach and the worst-case runtime for the probabilistic model checking of the computed causality classes. The worst-case runtime complexity of the qualitative causality checking approach, which was shown in Section 6.5, is RT(CausalityChecking) $\in \mathcal{O}(|t|^2)$ where $|t|$ is the number of traces. The worst-case runtime of one probabilistic model checking run is in $\mathcal{O}(|S|^3 + q * T_{max})$ where $|S|$ is the number of states in the Markov-chain, q is the uniformization rate and T_{max} is the maximal time bound of the probabilistic property, as was shown in [7]. Since in the worst-case there can be as many causality classes as traces in the model the worst-case runtime for the probabilistic model checking of the computed causality classes is in $\mathcal{O}(|t| * (|S|^3 + q * T_{max}))$. Consequently, the overall worst-case complexity of the combined approach is in $\mathcal{O}(|t|^2 + |t| * (|S|^3 + q * T_{max}))$.

The worst-case memory consumption of the combined approach is dominated by the qualitative causality checking which has a worst-case memory consumption that is in $\mathcal{O}(|t|)$ where $|t|$ is the number of traces.

8.6 Experimental Evaluation

In order to evaluate the performance of the proposed combined approach, we have extended the SpinCause tool that we used in Chapter 6. The following experiments were performed on a PC with two Intel Xeon Processors (4 cores with 3.60 Ghz) and 144 GBs of RAM. We evaluate the performance of the combined approach using the case studies presented Section 6.6. The PRISM models of the case studies are the

ones we manually created for Chapter 7, in practical usage scenarios the PRISM models can also be automatically synthesized from higher-level design models, as for instance by the QuantUM tool [72]. For the computation of the causality classes we used the iterative causality checking approach with parallel BFS (using 10 threads). For the experiments we set the mission time T of the analyzed properties to 1 hour.

8.6.1 Railroad Crossing

For the railroad crossing example the combined approach generates the same event order logic formula, characterizing the causal events for the hazard, as the qualitative and probabilistic causality checking approach. Furthermore, the probability values computed for the causality classes and the hazard are equal to the values computed with the probabilistic causality checking approach.

8.6.2 Airbag System

The event order logic formula, characterizing the causal events for the inadvertent deployment of the airbag, computed by the combined approach is equal to the EOL formula qualitative and probabilistic causality checking approach. Furthermore, the probability values computed for the causality classes and the hazard are equal to the values computed with the probabilistic causality checking approach.

8.6.3 Embedded Control System

The combined approach computes the same event order logic characterizing the causal events for a shutdown of the embedded system as the qualitative and probabilistic causality checking approach. Furthermore, the probability values computed for the causality classes and the hazard are equal to the values computed with the probabilistic causality checking approach.

8.6.4 Train Odometer Controller

The event order logic formula returned by the combined approach is Ψ = (Start_W_Fail_S $\wedge$ Wait_W_Fail_S) $\wedge$ ($\neg$failureDeteted $\wedge_]$ Wait_Mon_Fail) $\vee$ (Start_W_Fail_F $\wedge$ Wait_W_Fail_F) $\wedge$ ($\neg$failureDeteted $\wedge_]$ Wait_Mon_Fail) $\vee$ Wait_R_Fail $\wedge$ ($\neg$failureDeteted $\wedge_]$ Wait_Mon_Fail, which is equal to the EOL formula computed by the qualitative causality checking approach.

The total probability is $2.8232 \cdot 10^{-2}$ which can be broken down into:

- ((Start_W_Fail_S$\wedge$Wait_W_Fail_S)$\wedge$($\neg$failureDeteted$\wedge_]$Wait_Mon_Fail)) with a total probability of $1.5540 \cdot 10^{-2}$ and an exclusive probability of $8.0313 \cdot 10^{-3}$,
- ((Start_W_Fail_F $\wedge$ Wait_W_Fail_F) $\wedge$ ($\neg$failureDeteted $\wedge_]$ Wait_Mon_Fail)) with a total probability of $3.1482 \cdot 10^{-3}$ and an exclusive probability of $5.5040 \cdot 10^{-4}$, and

- (Wait_R_Fail $\wedge$ (¬failureDeteted $\wedge_{]}$ Wait_Mon_Fail) with a total probability of $1.9650 \cdot 10^{-2}$ and an exclusive probability of $9.5438 \cdot 10^{-3}$.

The probabilistic causality checking approach added two causality classes to the result, that combine the first and the third and the second and the third causality class. The exclusive probabilities of the causality classes computed by the combined approach are equal to the probabilities computed for the same causality classes by the probabilistic causality checking approach.

8.6.5 Airport Surveillance Radar

For both variants of the airport surveillance radar (ASR) the event order logic formula returned by the combined approach, characterizing the causal information for loosing the flight path information of an aircraft, is Ψ = (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ PSR_only $\wedge$ coastTrack) $\vee$ (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ SSR_only $\wedge$ coastTrack). This EOL formula is equal to the EOL formula computed by the qualitative causality checking approach. Since the probabilistic causality checking approach was not able to produce a result for the ASR case study, we have no result of the probabilistic causality checking approach with which we could compare this result. The redundancy in the 2 channel variant of the ASR does not have an impact on the probability of the case where the position information of an aircraft is lost, since the redundant channel is only added in order to increase the availability. Note that the used probability values are mock numbers, since the real values are intellectual property of our industrial partner. The total probability for loosing the position information of an aircraft within one hour is $8.9246 \cdot 10^{-10}$ which can be broken down into:

- (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ PSR_only $\wedge$ coastTrack) with a total and exclusive probability of $8.8550 \cdot 10^{-10}$ and
- (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ SSR_only $\wedge$ coastTrack) with a total and exclusive probability of $6.9625 \cdot 10^{-12}$.

8.6.6 Discussion

	Combined Approach		Probabilistic Causality Checking	
	Runtime (sec.)	Memory (MB)	Runtime (sec.)	Memory (MB)
Railroad	0.80	18.11	2.00	10.00
Airbag	2.08	18.86	609.00	389.00
Embedded	0.91	18.75	1,936.00	383.00
Train Odometer	5.88	19.53	17,154.00	1,739.00
ASR 1 Channel	106.40	238.97	out of mem.	out of mem.
ASR 2 Channel	38,605.02	7,728.63	out of mem.	out of mem.

Table 8.1: Runtime and memory consumption of the combined approach and the probabilistic causality checking approach.

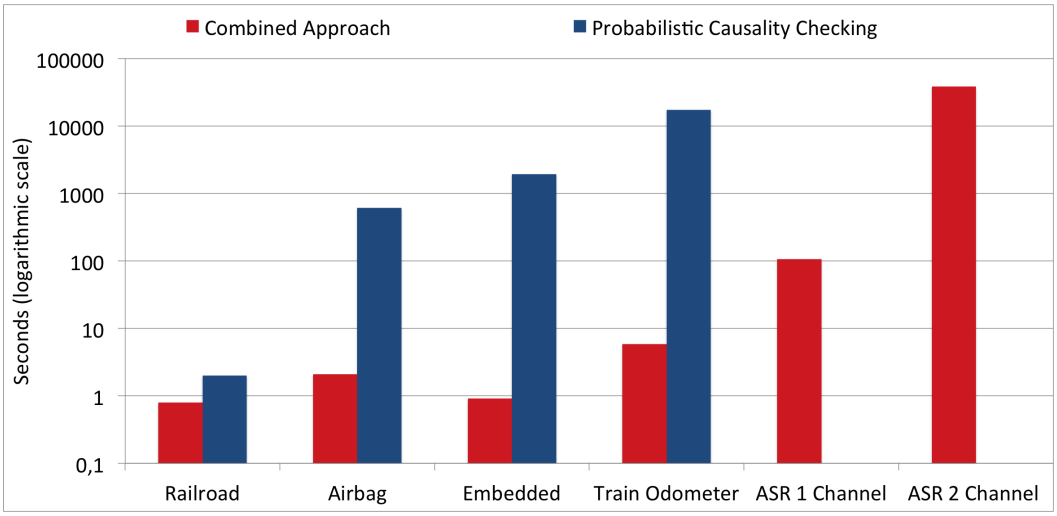


Figure 8.1: Runtime needed for the combined approach and the probabilistic causality checking approach (logarithmic scale).

As we would expect, for all case studies the total probability returned by the combined approach is equal to the probability returned for the respective probabilistic property by PRISM after a probabilistic model checking run. If we sum up the probabilities of the traces computed by DiPro for each causality class and only consider traces that belong to exactly one causality class, then the sum of the probability of each causality class is equal to the corresponding exclusive probability (p_{ψ_excl}) value of that causality class computed by the combined approach. If, on the other hand, we sum up the probabilities of the traces computed by DiPro for each causality class and also consider the probability mass of traces that belong to more than one causality class, the probability sum of each causality class is equal to the corresponding total probability (p_{ψ}) value of that causality class computed by the combined approach.

Table 8.1 shows the runtime and memory consumption of the combined approach and the probabilistic causality checking approach for each of the case studies. The different runtime values are visualized in Figure 8.1 and the memory consumption is visualized in Figure 8.2. The runtime and memory values for the combined approach include the runtime and memory needed for all steps of the approach, namely translation from PRISM to Promela, causality checking, alternating automata derivation and mapping to PRISM, and the PRISM model checking.

The combined approach consumes significantly less run time and memory than the probabilistic causality checking approach. This difference can be explained by the fact that for the probabilistic causality approach the probability of each trace in the counterexample needs to be computed individually, which requires a probabilistic model checking of a part of the model for each trace. The combined approach reduces the number of probabilistic model checking runs to the number of the computed causality classes. The relatively low runtime that is needed by

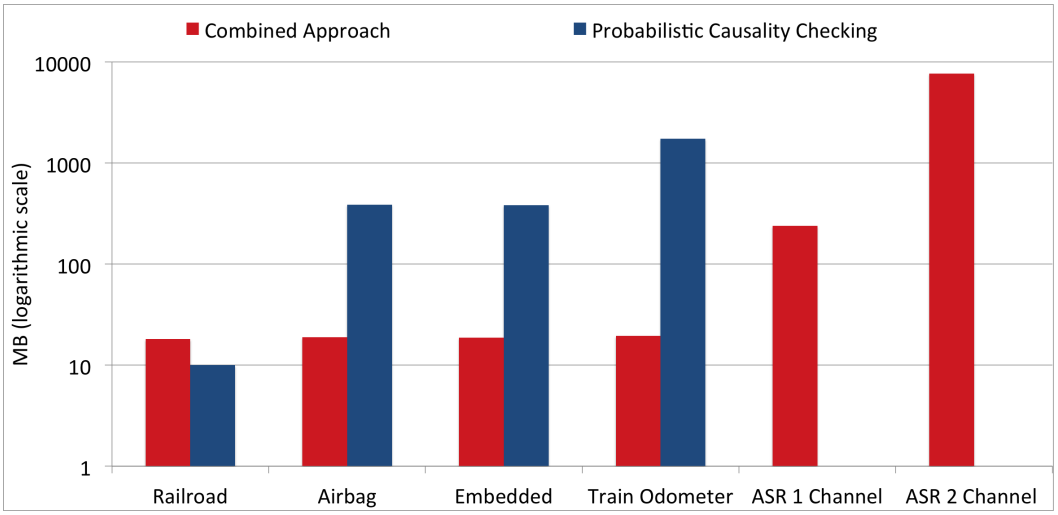


Figure 8.2: Runtime consumption of the combined approach and the probabilistic causality checking approach (logarithmic scale).

the combined approach for the embedded case study as compared to the other case studies can be explained by the relatively short length of the traces in the causality classes of the embedded case study.

8.6.7 Summary

We have evaluated the usefulness and performance of the proposed combined causality checking approaches on 5 case studies. Two of the case studies are taken from industrial projects. The experiments clearly show that the combined causality checking approach significantly consumes less runtime and memory as the probabilistic causality checking approach. With the combined causality checking approach it is possible to analyze the ASR case study and to compute the probabilities of the causality classes which was not possible with the probabilistic causality checking approach.

Chapter 9

Causality Checking at the Limits of Scalability

The content of this chapter is have not been previously published.

Contents

9.1 Introduction

The causality checking approaches described in Chapter 6, Chapter 7 and Chapter 8 are based on an exhaustive exploration of the state-space. One major limitation of all methods using state-space exploration techniques, including explicit state model-checking, is that the runtime and memory consumption of these methods is mainly influenced by the number of states in the model. It can be the case that the number of states in a model is extremely large and thus the state-space exploration either runs out of available memory or the required runtime is too high to be acceptable. In literature this problem is referred to as the *state-space explosion problem* [8].

With the approaches described in Chapter 6 and Chapter 8 it is possible to perform qualitative and probabilistic causality checking for all of the case studies that we have presented in this thesis. Nevertheless, we discuss the implications of an incomplete state-space exploration on the completeness and soundness of the causality checking results in Section 9.2. Furthermore, we discuss in Section 9.2.3 how partial order reduction [8], a technique used in explicit state-model checking to reduce the number of possible event orderings that have to be analyzed, affects the causality checking results. In Section 9.3 we propose strategies to increase the scalability for practical application scenarios.

9.2 Causality Checking and Incomplete State-Space Exploration

In the experiments described in Chapter 6 and Chapter 8 we were able to fully explore the state-space of the model and, therefore, obtained a complete and sound causality checking result. In this section we discuss the implications if an exhaustive state-space exploration is no longer possible.

9.2.1 Completeness

In case the state-space of a model is infinite or has not been fully explored, for example due to an out of memory error, we have no guarantee that all bad traces have been found, since there always might be a trace leading to a bad state in the part of the state-space that we have not yet explored. Theorem 43 in Section 5.3 requires that all bad execution traces are found in order for the causality checking result to be complete. Consequently, if no full state-space exploration was performed the result of the causality checking is potentially incomplete.

It remains to be discussed whether although no complete state-space exploration was performed we can make any assumption on the result generated by the causality checker. Since the traces found during an incomplete state-space exploration depend on whether BFS or DFS is used, we need to discuss both cases separately.

BFS explores the state-space in an exploration order that leads to a monotonically increasing length of the execution traces. Consequently, if BFS is used we know that all traces with length n up to the current search depth d have been found.

Theorem 59. *If BFS explored the state-space up to search depth* d *all causality classes with the length of* n *events and* $n \leq d$ *have been found and all possible event orderings have been computed for these causality classes.*

Proof. Assume that there exists a causality class with the length of n events and $n \leq d$, where d is the current search depth of BFS, that was not yet found. By definition of the causality class there exists at least one trace with length n which would be represented by this causality class. Due to the minimality constraint no bad trace with length $n' < n$ exists. For the causality class not to be found it would be required that the trace with length n, which would be represented by the causality class, was not found by BFS. Since $n \leq d$ holds the trace is found by BFS and, consequently, all causality classes with the length of n events and $n \leq d$ are found. □

The literature on bounded model checking [19, 20] shows that an upper bound for the search depth that is needed in order to guarantee the completeness of the bounded model checking can be given. This upper bound is also called completeness threshold. The upper bound for non-reachability properties is the diameter or radius r, which is the minimal number of steps required for reaching all states.

Definition 40. *Radius (r). The radius of some transition system T is defined as:* $r(T) = min\{i \mid \forall t \in \mathcal{S}.\ \exists j : j \leq i.\ R(t) \rightarrow R(t,j)\}$*, where* $R(t)$ *is true whenever* t *is reachable from the initial state* s_0 *and* $R(t,j)$ *is true whenever the state* t *is reachable from the initial state* s_0 *within* j *steps.*

If the state-space of the model is fully explored with BFS up to the search depth that is specified by the radius, it can be guaranteed that for each possible bad state the shortest trace leading into this bad state has been found. This is not sufficient in order to guarantee the completeness of the result of the causality checker, since we have not found all bad traces. Consider the following example, assume there are only two traces in the transition system T, namely, $\sigma = s_0$ a s_1 b s_2 and $\sigma' = s_0$ a s_1 c s_3 d s_2. Since s_2 is reachable from s_0 via s_1 within 2 steps and s_3 is reachable from s_0 via s_1 within 2 steps, the radius of T is $r = 2$. If now the search depth would be limited to $d = 2$ the trace $\sigma' = s_0$ a s_1 c s_3 d s_2 would never be found and we would potentially miss a causality class if σ' is a bad trace.

Another upper bound that is used in the literature on bounded model checking [19, 20] to guarantee the completeness of the bounded model checking is the recurrence diameter, which is the longest loop-free path between two states.

Definition 41. *Recurrence Diameter (rd). The recurrence diameter (rd) of some transition system T is defined as the minimal number* rd *with the following property. For every sequence of states* $s_0, ..., s_{rd+1}$ *starting from the initial state* s_0 *with* $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ *for* $i \leq rd$*, there exists* $j \leq rd$ *such that* $s_{rd+1} = s_j$*.*

If BFS is used to explore the state-space up to search depth $d = rd$ we know that we have found the longest loop-free traces to all bad states and according to Theorem 59 we know that we have found all causality classes with the length of n events and $n \leq d$. We also know that each bad trace σ that we have not yet found contains a loop and, consequently, we know that we already have found a trace σ' for which $\sigma' \subset \sigma$ holds.

Theorem 60. *The recurrence diameter* rd *of a transition system is a completeness threshold for causality checking with BFS.*

Proof. Assume that there exists a causality class ψ with the length of n events and $n > rd$ that was not found. By definition of the causality class there exists at least one trace σ with length n which would be represented by this causality class. For a trace σ to be longer than the recurrence diameter, it has to be longer than the longest loop-free trace and, consequently, contains a loop. For the trace σ leading to some bad state s_j there has to exist at least one loop-free trace σ' leading to s_j. Since σ contains a loop and σ' does not contain this loop $\sigma' \subset \sigma$ holds and since the trace σ' exists there also exists a causality class ψ' for which $\sigma' \vDash_e \psi'$ and $\sigma \vDash \psi'$ holds. This would violate the AC3 condition for ψ since ψ' is a subset of ψ. Consequently, no causality class with the length of n events and $n > rd$ can exist and the recurrence diameter is a completeness threshold for causality checking. □

In practice BFS does not explore a trace further if a loop was detected, since the state that where the loop is closed, already has been explored and is stored in the state-space V. This means that with BFS no trace that is longer than the recurrence diameter of the transition system will be found. Consequently, exploring the state-space to a search depth that is equal to the recurrence diameter is equal to performing a full state-space exploration with BFS. Nevertheless, it is possible that traces that are longer than the recurrence diameter are generated by the duplicate state matching method. These traces can be safely ignored without having an impact on the completeness of the causality checking result.

If DFS is used no assumptions on the length of the unknown traces can be made and, consequently, we can not make any assumptions on the length of the causality classes found during an incomplete state-space exploration with DFS.

9.2.2 Soundness

While an incomplete state-space exploration does not have an effect on the soundness of the counterexamples that are returned by DFS and BFS, it does have some effects on the soundness of the causality classes. Theorem 44 in Section 5.3 requires that all good and bad execution traces are found in order for the causality classes computed by the causality checker to be sound.

For a causality class not to be sound, there has to exist a good trace σ satisfying the causality class ψ which is a disjunct of the EOL formula Ψ returned by the causality checker. In the proof of Theorem 44 we showed that this can not happen if all bad and good traces are found, if this assumption no longer holds this has the following impact. ψ is added to Ψ because AC1-AC3 are satisfied for ψ. If $\sigma \vDash \psi$ is true, the event combination specified by ψ occurs on σ. For σ being a good trace, this would require that the property violation is prevented by some event on σ which is not constrained by ψ. If such an event exists, and the good trace is found during the state-space exploration the AC2(2) test fails and ψ is not added to Ψ, but instead an altered version of ψ ensuring the non-occurrence of the event preventing the property violation is added to Ψ. Since on σ at least one event preventing the property violation occurs, $n < n'$ holds where n is the number of events in the causality class and n' is the number of events occurring on σ. Consequently, it is possible that the causal non-occurrence of events is not detected if the causality class is found but the trace σ required to detect the causal non-occurrence is not found due to the incomplete state-space exploration.

The detection of the causal non-occurrence of events can not be guaranteed if the state-space exploration is incomplete, regardless of whether DFS or BFS is used, since it can always be the case that there exists a good trace that is not found due to the incomplete state-space exploration and that violates the AC2(2) condition for some causality class.

If BFS is used it is guaranteed that all traces with a length up to $d-1$, where d is the current search depth, have been found. Consequently, we can guarantee that for a causality class with length of n events and the current search depth d, the causality

class is sound, taking into account that there might exist a combination of at least $k = (d+1) - n$ events which can prevent the property violation. With an increasing k the likelihood of the existence of such an event combination is decreasing.

Theorem 61. *If BFS explored the state-space up to search depth* d *all causality classes that have been found are sound, taking into account that an event combination with at least* $k = (d+1) - n$ *events, where* n *is the number of events in the causality class, voiding the soundness of the causality class could exist.*

Proof. If BFS is used it is guaranteed that all traces with a length of up to $d-1$, where d is the current search depth, have been found. According to Theorem 59 all causality classes with the length of n events and $n \leq d-1$ have been found. Since all good traces up to length $d-1$ are found, the causal non-occurrence of event combinations with length up to $k = (d-1) - n$ can be detected for each causality class with length n of events. Since there potentially exists a good trace with length $n' \geq d-1$, which is not found due to the incomplete state-space exploration, there potentially exists an event combination with at least $k = (d+1) - n$ events for which the non-occurrence is causal and that is not detected by the algorithm. Consequently, if BFS explored the state-space up to search depth d all causality classes that have been found are sound taken into account that an event combination with at least $k = (d+1) - n$ events, where n is the number of events in the causality class, voiding the soundness of the causality class could exist. □

In Section 9.2.1 we have shown that the recurrence diameter of the transition system is a completeness threshold for causality checking. This completeness threshold ensures that all loop-free traces that exist in the transition system are found. The recurrence diameter can also be used to compute an upper bound of the events that can potentially void the soundness of a causality class.

Theorem 62. *The maximal number of events k in an event combination that can potentially void the soundness of a causality class is $k = rd - n$, where rd is the recurrence distance and n is the length of the causality class.*

Proof. For a combination of events to void the soundness of a causality class ψ with the length n of events, there has to exists a good trace σ' with length $n' > n$ and $\sigma' \vDash_e \psi$. Suppose that the length of σ' is $n' > rd$, which means that sigma contains a loop. Then there exists another good trace σ'' with length $n'' = rd$ and $n'' < n'$ for which $\sigma'' \subset \sigma'$ holds. Since σ'' is a good trace and $\sigma'' \subset \sigma'$ holds, σ'' already contains all events that are responsible for the prevention of the property violation and thus need to be considered by the causal non-occurrence test. Consequently, the maximal number of events that can potentially void the soundness of a causality class is $k = rd - n$, where rd is the recurrence distance and n is the length of the causality class. □

From Theorem 62 it follows that if we have explored the state-space with BFS up to the search depth $d = rd$ we are able to detect all events for which their

non-occurrence is causal. This means that the recurrence diameter is a soundness threshold for causality checking with BFS.

Theorem 63. *The recurrence diameter* rd *of a transition system is a soundness threshold for causality checking with BFS.*

Proof. According to Theorem 62 the maximal number of events k in a event combination that can potentially void the soundness of a causality class is $k = rd - n$, where rd is the recurrence distance and n is the length of the causality class. If the state-space is explored with BFS up to the search depth $d = rd$ we know that all traces with a length up to rd are found. Consequently, we are able to detect all events for which their non-occurrence is causal and the recurrence diameter rd of a transition system is a soundness threshold for causality checking with BFS. □

Exploring the state-space to a search depth that is equal to the recurrence diameter is equal to performing a full state-space exploration with BFS, as we have argued in the previous Section. Nevertheless, it is possible that traces that are longer than the recurrence diameter are generated by the duplicate state matching method. These traces can be safely ignored without having an impact on the soundness of the causality checking result.

For an incomplete state-space exploration with DFS no assumption on the length of the traces that are not found can be made. Consequently, the soundness of the causality classes computed using an incomplete state-space exploration with DFS can not be guaranteed.

9.2.3 Causality Checking and Partial Order Reduction

Partial order reduction [8] is a technique used in explicit state-model checking to reduce the number of possible event orderings that have to be analyzed. Partial order reduction leverages the fact that under some circumstances not all possible interleavings of concurrently executed processes need to be explored, since it might be sufficient to explore one interleaving if the order of the events does not have an effect on the analyzed property. The order of two events does not have an effect on the analyzed property, if regardless of the order in which they are executed, the same state is reached. Obviously, if the number of interleavings to be analyzed can be reduced to one or a few representatives, this reduces the state-space significantly.

The question is whether partial order reduction has an effect on the results generated by the causality checker. For the causality checking to provide a complete and sound result, it is required that all possible good and bad traces are identified. Partial order reduction represents a number of interleavings with one or a few representatives. Therefore, not all traces are identified if partial order reduction is used. Since at least one representative of the equivalence of interleavings is found, all possible event combinations are found, but since not all possible interleavings are identified the causality checker is not able to correctly identify the causal event orderings. Consequently, if partial order reduction is used during causality checking,

the returned causality classes might not cover all possible orderings and thus might not cover all bad execution traces. As a result, the causality checking result is not complete if partial order reduction is used. The soundness of the causality checking result is not affected by the causality checking result. The reason for that is that for each representative of a bad trace we also find at least one representative of a good super trace, if such a good super trace exists. Therefore, the causal non-occurrence of events will be detected correctly if partial order reduction is used.

Nevertheless, it is possible to use partial order reduction during the first iteration of the iterative causality checking approaches proposed in Chapter 6, since the first iteration aims at identifying the causality classes without identifying the causal event orderings.

The partial order reduction implemented in the SpinJa model checker [33] did not lead to a reduction of the state-space for the models we have analyzed. The reason for this is that the partial order implementation in SpinJa requires for interleavings to be pruned that they only access local variables of a process, but since the case studies used in this paper only use globally defined variables no interleaving meets this requirement.

In future work we will investigate whether the information obtained by the static analysis performed in order to do partial order reduction can be used to identify all causal event orders even if partial order reduction is used.

9.2.4 Summary and Implications for Practical Usage Scenarios

In conclusion, if DFS, BFS or one of the iterative causality checking approaches is used and the state-space is not fully explored by the causality checker it is possible that not all causality classes are found and that the found causality classes do not take all possible causal non-occurrences of events into account. For the algorithms based on BFS we can guarantee the completeness and soundness with respect to the current search depth d.

We have shown that the recurrence diameter of a transition system can be used as a completeness and soundness threshold for causality checking and it was shown in [58] that the recurrence diameter can be efficiently computed. But even if the completeness and soundness of the causality checking results can not be guaranteed in the event of an incomplete state-space exploration, we believe that the information obtained by an incomplete state-space exploration is still valuable for the debugging of the system, since the causality classes provide information about the event combinations that are involved in violating a property. The results obtained by causality checking using an incomplete state-space exploration with BFS can, due to Theorem 59 and Theorem 61 be used in the safety analysis of a system. The longer a causality class is, the more events have to jointly occur in order for the property violation to occur. In practical use-case scenarios the interesting causality classes are the short causality classes because this means that a low number of events, e.g., hardware failures, suffice to violate the property. For this reason engineers only consider a limited number of failure events that can jointly cause

a property violation during the safety analysis. Such a failure of the system, that requires more than one failure event, is also called multiple-point failure [55]. The ISO 26262 [55] safety standard for automotive systems, for instance, restricts the analysis in most cases to dual-point failures, which are multiple-point failures caused by two failure events. Consequently, limiting the completeness of the causality class to some search depth d is acceptable in practical use case scenarios. The limitation of the soundness of the causality checking results to some search depth d might lead to causality classes where the causal non-occurrence of some events was not detected. In practical use case scenarios the events, where the non-occurrence is causal, are recovery and repair-mechanisms. Since the main objective of the safety analysis is to detect event combinations that can lead to a property violation, the information that there exists a combination of events, described by a causality class, that can lead to a property violation is helpful for the engineer, even if there might be additional events that if they occur jointly with the events described by the causality class can prevent the property violation. Consequently, limiting soundness of the causality checking results to some search depth d is acceptable in practical use case scenarios.

In Section 9.3 we discuss strategies that can be applied in order to increase the scalability of the causality checking approach. The proposed strategies leverage the fact that in practical use case scenarios it is acceptable that when the state-space is not fully explored by BFS we can guarantee the completeness and soundness with respect to the current search depth d.

9.3 Strategies to Increase Scalability

In this section we discuss strategies that can be applied if the causality checker runs out of memory during a full state-space exploration. Since the iterative approach using parallel BFS offers the best scalability of all approaches discussed in Chapter 6 we base our strategies to increase the scalability on this variant. In theory the proposed strategies can be applied to all BFS based approaches.

9.3.1 Strategy 1: Limiting the Search Depth

The first strategy allows to specify an upper bound k for the number of events in a causality class. The state-space exploration is aborted after the maximal search depth k has been fully explored. Obviously aborting the search at depth k is an incomplete exploration of the state-space and the completeness and soundness of the causality checking result can only be guaranteed with respect to the search depth k.

We use the case studies from Section 6.6 to evaluate how the runtime and memory of the iterative causality checking approach using parallel BFS (with 10 threads) changes if we limit the maximum number k of events in a causality class to 15 and 20 events, respectively. The following experiments were performed on a PC with two Intel Xeon Processors (4 cores with 3.60 Ghz) and 144 GBs of RAM.

	Iterative Approach with Parallel BFS (k = 15)		Iterative Approach with Parallel BFS (k = 20)		Iterative Approach with Parallel BFS (complete exploration)	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
Railroad	0.74	17.92	0.74	17.92	0.74	17.92
Airbag	1.59	18.51	1.59	18.51	1.59	18.51
Embedded	0.75	17.99	0.75	17.99	0.75	17.99
Train Odometer	1.44	19.11	1.44	19.11	1.44	19.11
ASR 1 Channel	8.61	37.31	10.10	50.76	50.37	195.51
ASR 2 Channel	51.91	458.15	94.80	826.73	1,101.99	6,967.00

Table 9.1: Runtime and memory needed for the iterative causality checking with parallel BFS for k = 15, k = 20 and a complete exploration.

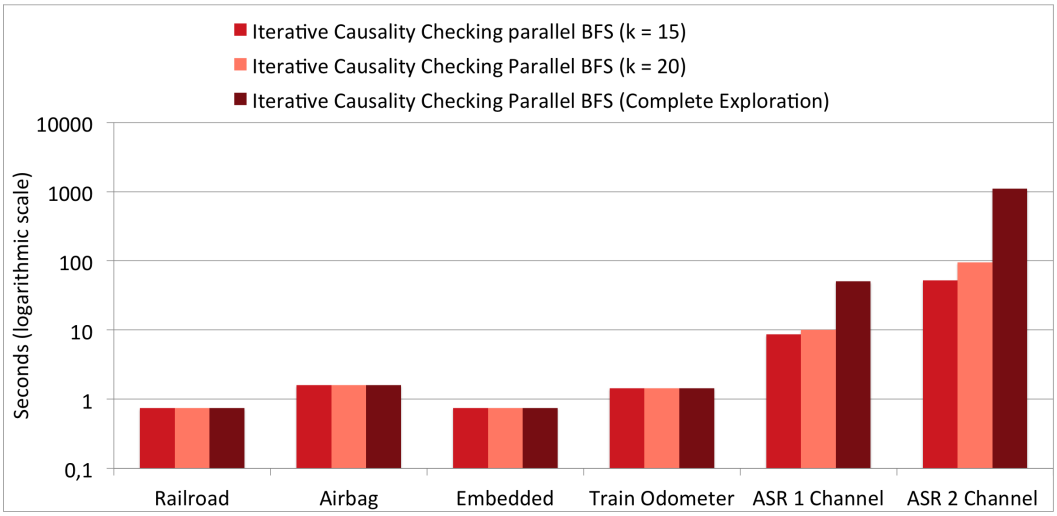


Figure 9.1: Runtime needed for the iterative causality checking with parallel BFS for k = 15, k = 20 and a complete exploration (logarithmic scale).

Table 9.1 shows the runtime and memory needed for the iterative causality checking with parallel BFS for k = 15, k = 20 and a complete exploration. The different runtime values are visualized in Figure 9.1 and the memory consumption is visualized in Figure 9.2.

The EOL formulas returned by the causality checking runs with limited depth are in all cases equal to the EOL formulas that were generated by a complete exploration. This means that no causality class is missing and the events for which their non-occurrence is causal have been found.

For the railroad, airbag, embedded, and train odometer case studies the depth limit does not have an effect on the runtime or memory consumption, since the maximal reachable depth of a complete exploration is less than the depth limit k. For the ASR 1 channel and ASR 2 channel variant the depth limit leads to a decrease in both runtime and memory, while providing the same results. The reason

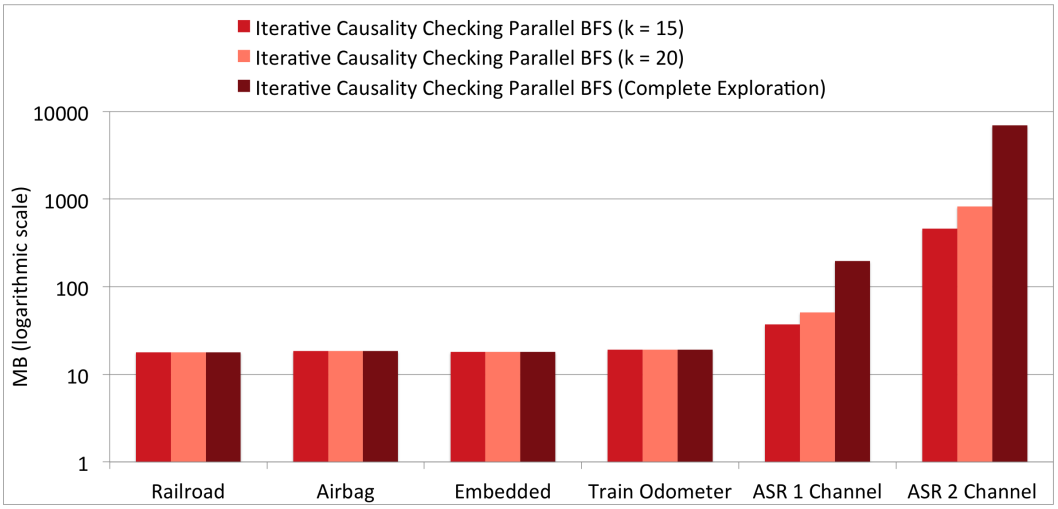


Figure 9.2: Memory needed for the iterative causality checking with parallel BFS for k = 15, k = 20 and a complete exploration (logarithmic scale).

for the reduction of runtime and memory is that not the full state-space has to be explored and, consequently, less traces have to be processed.

The disadvantage of this approach is that we can only guarantee the completeness and soundness of the causality classes with respect to the search depth k. If, for instance, a very small value for k is selected it is possible that no causality classes can be computed or that some causality classes are missed since the state-space is not fully explored. The strategy presented in the next Section extends this strategy by providing an estimate of the probability sum of all missed causality classes.

9.3.2 Strategy 2: Estimation of the Residual Probability

If the causality checking is aborted at search depth k, it is possible that some causality classes are missed, since not the full state-space is explored. In order to get an estimate whether any meaningful information is missing, we can compute an estimate of the probability sum of all missing causality classes.

In order to compute the estimate we perform the following steps:

- We compute the causality classes up to search depth k with the strategy proposed in Section 9.3.1 and use the combined approach proposed in Section 8 in order to compute the probabilities for the causality classes.
- We compute the total probability p_{total} for the property violation with PRISM.
- We compute the probability of the causality classes of the computed EOL formula $\Psi = \psi_1 \vee ... \vee \psi_n$ by computing the probability p_{result} for the CSL formula $\mathcal{P}_{=?}[(true)U(\text{acc\_}\psi_1|...|\text{acc\_}\psi_n)]$ with PRISM.
- The probability $p_{\text{missing}} = p_{\text{total}} - p_{\text{result}}$ is the probability of the missing causality classes.

	Combined Approach (k = 15)		Combined Approach (k = 20)		Combined Approach (complete exploration)	
	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)	RT (sec.)	Mem (MB)
ASR 1 Channel	82.61	37.88	84.94	51.34	106.40	238.97
ASR 2 Channel	37,768.02	458.67	37,956.00	1,913.23	38,605.02	7,728.63

Table 9.2: Runtime and memory needed for the combined approach for k = 15, k = 20 and a complete exploration.

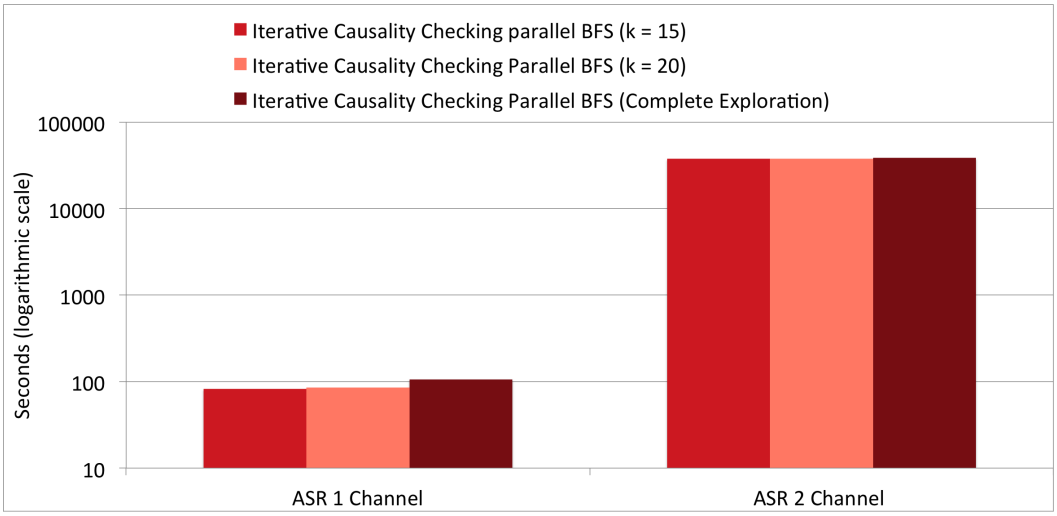


Figure 9.3: Runtime needed for the combined approach with k = 15, k = 20 and a complete exploration (logarithmic scale).

Table 9.2 shows the runtime and memory needed for the combined approach with k = 15, k = 20 and a complete exploration. The different runtime values are visualized in Figure 9.3 and the memory consumption is visualized in Figure 9.4. Since we have already seen in Section 9.3.1 that for the railroad crossing example, the airbag system, the embedded control system, and the train odometer case studies the depth limit does not have an effect on the runtime or memory consumption, since the maximal reachable depth of a complete exploration is less than the depth limit, we only show the experimental results for the ASR case studies. For both the 1 channel variant and the 2 channel variant of the ASR and k = 15 and k = 20 the probability of potentially missed causality classes is zero, consequently, all causality classes have been computed. For both variants the consumed runtime and memory of the combined approach with k = 15 and k = 20 is less than the runtime and memory consumed by the combined approach with a full exploration.

9.3.3 Summary

The experiments we have performed in Section 9.3.1 and Section 9.3.2 show that by limiting the analysis depth the runtime and memory that is needed for the causality

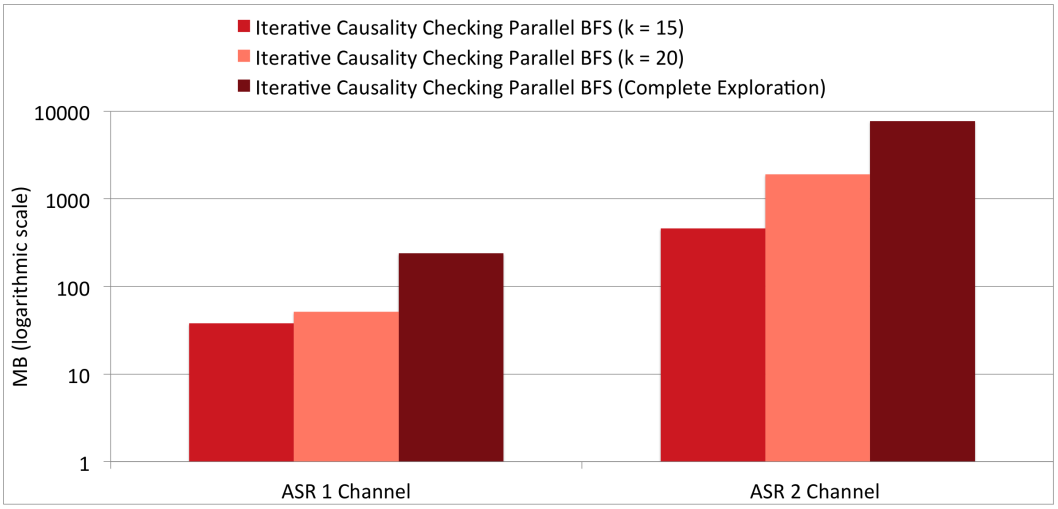


Figure 9.4: Memory needed for the combined approach with k = 15, k = 20 and a complete exploration (logarithmic scale).

checking of large models can be reduced. We have argued in Section 9.2.4 that in practical use case scenarios it is acceptable that the completeness and soundness of the causality checking can only be guaranteed with respect to the current search depth d, if the state-space can not be fully explored with BFS. For the experiments in Section 9.3.1 and Section 9.3.2 we obtained the same causality classes as we have obtained using a full exploration, consequently, no information was lost by limiting the search depth. Even if one or more causality classes would be missed by limiting the analysis depth, the strategy described in Section 9.3.2 allows for the computation of the probability of the unknown causality classes. In practical use case scenarios this can be used to incrementally increase the search depth k until either all causality classes are found and the probability of the potentially missed causality classes is zero, or until the probability of the potentially missed causality classes is below some predefined threshold.

Chapter 10

Causality Checking and Fault Trees

The content of this chapter is based on the publications [61, 62, 78].

Contents

10.1 Introduction

In order to aid the adoption of causality checking in industrial use case scenarios it is a necessity to represent the causality checking results in a concise and easy to understand form. Furthermore, it is necessary to discuss how the results obtained with causality checking compare with the results obtained by traditional safety analysis methods.

In Section 10.2 we show how event order logic formulas that have been generated by the causality checker can be represented by fault trees [97], a method used in industry to reason about causal relationships between property violations and events. Since fault trees are not expressive enough to represent all the order constraints that can be specified using the event order logic, we propose an additional graphical representation of the event order logic order constraints in Section 10.3. The relationship of causality classes generated by causality checking and the minimal cut sets used in manual fault tree analysis is discussed in Section 10.4. In Section 10.5 we discuss how root-causes can be identified using causality checking.

10.2 Mapping of Event Order Logic Formulas to Fault Trees

The event order logic formula returned by the causality checker is a concise and compact representation of the causal event combinations. Nevertheless the inter-

pretation of an event order logic formula requires knowledge about the syntax and semantics of the event order logic. In this section we show how the event order logic formulas generated by the causality checker can be graphically represented by fault trees, a representation that engineers are commonly familiar with. The algorithm that translates EOL formulas to fault trees is given in Listing 10.1 and implements the following graphical representation of EOL formulas to fault trees.

Definition 42. *Graphical Representation of EOL formulas with Fault Trees. Let* Ψ *an event order logic formula returned by the causality checker.* Ψ *can be represented as a fault tree by applying the following mapping:*

- *An* intermediate event *is used in order to represent the top-level event representing the property violation. (Listing 10.1, line 4)*
- *If there is only one causality class in* Ψ*, the fault tree mapping for this causality class is directly connected to the top-level event (Listing 10.1, lines 6-9). If there is more than one one causality class in* Ψ*, the top-level event is connected to an* OR*-gate which connects the different causality classes of the event order logic formula* Ψ *(Listing 10.1, lines 10-18).*
- *Each causality class is connected to the* OR*-gate by an* intermediate event*, which is then connected to an* AND*- or* PAND*-gate connecting the events in that causality class (Listing 10.1, lines 22-54).*
  - *If a causality class consists of only one event the* AND*-gate is omitted and the event is directly connected to the* intermediate event *(Listing 10.1, lines 25-28).*
  - *If a causality class contains at least one ordered EOL-operator it is represented using an* PAND*-gate, else with an* AND*-gate (Listing 10.1, lines 31-42).*
  - *Since the* PAND*-gate is not expressive enough to express the event orderings that can be expressed with the event order logic, a constraint containing the event order logic formula of the causality class is added to the* PAND*-gate (Listing 10.1, line 40).*
  - *The events of the causality classes are represented by* basic events *that are connected to the* PAND*-gate or* AND*-gate (Listing 10.1, lines 43-51).*

```
1 function EOL2FT(Ψ)
2 {
3     FaulTree ft;
4     ft.TLE = new IntermediateEvent('φ');
5
6     IF( Number of ψ in Ψ == 1)
7     {
8         ft.TLE.add(SUB_FT_FOR_CC(ψ));
```

```
 9       }
10       ELSE
11       {
12            ORGate or;
13            FOR EACH Causality Class ψ in Ψ
14            {
15                 or.add(SUB_FT_FOR_CC(ψ));
16            }
17            ft.TLE.add(or);
18       }
19       return ft;
20  }
21
22  function SUB_FT_FOR_CC(ψ)
23  {
24       IntermediateEvent ie;
25       IF( Number of Events a in ψ == 1)
26       {
27            ie.add(a);
28       }
29       ELSE
30       {
31            IF( ψ contains ∧ OR
32                 ψ contains ∧] OR
33                 (ψ contains ∧< AND ψ contains ∧>))
34            {
35                 PANDGate pand;
36                 FOR EACH Event a in ψ
37                 {
38                      pand.add(a);
39                 }
40                 pand.OC = ψ;
41                 ie.add(pand);
42            }
43            ELSE
44            {
45                 ANDGate and;
46                 FOR EACH Event a in ψ
47                 {
48                      and.add(a);
49                 }
50                 ie.add(and);
51            }
52       }
53       return ie;
54  }
```

Listing 10.1: Pseudocode of the EOL formula to fault tree translation algorithm

Example 4. *Figure 10.1 shows how the EOL formulas $a \wedge b$ and $a \owedge b$ can be represented by fault trees.*

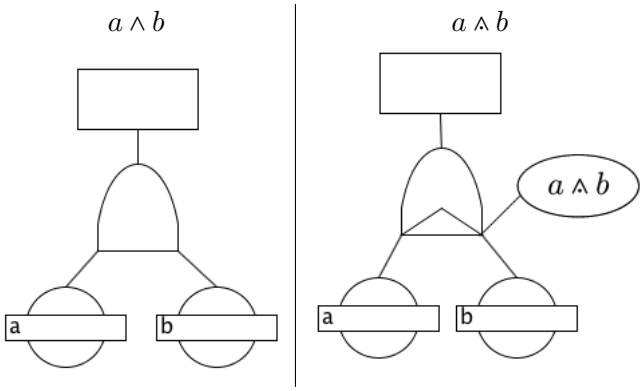


Figure 10.1: Example mappings of EOL formula to fault trees.

In the following Sections we show the fault trees for the case studies presented in Section 6.6. The fault trees are annotated with the probabilities computed with the combined approach presented in Chapter 8. Since the mapping from EOL formulas to fault trees is a mere rewriting step, the runtime and memory consumption introduced by this step is constant and can be neglected, in fact the additional runtime and memory needed is so small that it was not possible to measure the runtime and memory that was added by the fault tree mapping in our experiments.

10.2.1 Railroad Crossing

Figure 10.2 shows the fault tree of the railroad crossing running example described in Section 3.2. For better readability we have replaced the order constraints of the *PAND*-gates by the identifiers OC1 and OC2, where

- OC1 = $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \owedge \mathrm{Cc})) \owedge_{<} \neg\mathrm{Cl} \owedge_{>} \mathrm{Tc})$ and
- OC2 = $(\mathrm{Ta} \wedge (\mathrm{Ca} \owedge \mathrm{Cc})) \owedge_{<} \neg\mathrm{Cl} \owedge_{>} (\mathrm{Gc} \wedge \mathrm{Tc})$.

10.2.2 Airbag System

The fault tree for the airbag system described in Section 6.6.2 is shown in Figure 10.3. For better readability we have replaced the order constraints of the *PAND*-gates by the identifiers OC1, OC2, OC3 and OC4, where

- OC1 = (FETStuckHigh ⩓ FASICStuckHigh),
- OC2 = (MicroControllerFailure ⩓ enableFET ⩓ FASICStuckHigh),
- OC3 = (FETStuckHigh ⩓ MicroControllerFailure ⩓ armFASIC ⩓ fireFASIC), and
- OC4 = (MicroControllerFailure ⩓ enableFET ⩓ armFASIC ⩓ fireFASIC).

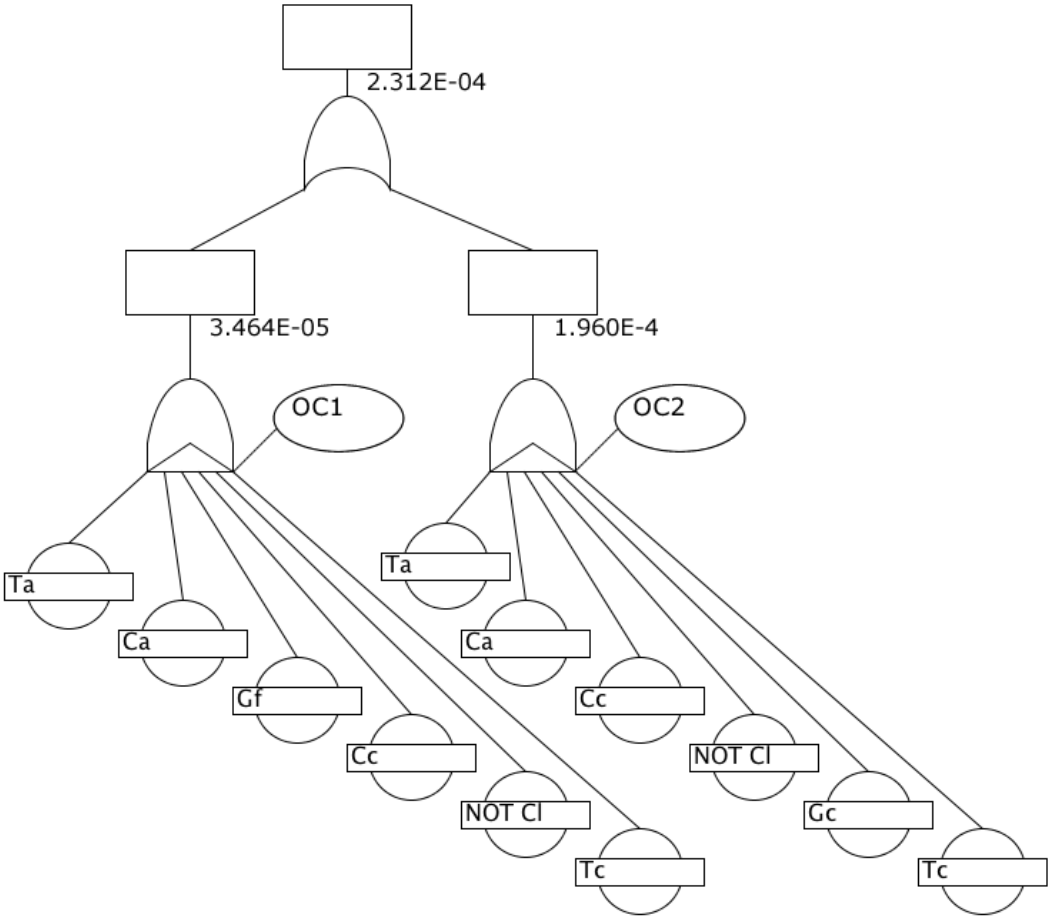


Figure 10.2: Fault tree of the railroad crossing running example.

10.2.3 Embedded Control System

Figure 10.4 shows the fault tree of the embedded control system presented in Section 6.6.3. For better readability we have replaced the order constraints of the *PAND*-gates by the identifiers OC1 and OC2, where

- OC1 = (SensorFailure $\wedge$ SensorFailure) and
- OC2 = (ActuatorFailure $\wedge$ ActuatorFailure).

10.2.4 Train Odometer Controller

The fault tree of the train odometer controller introduced in Section 6.6.4 is shown in Figure 10.5. For better readability we have replaced the order constraints of the *PAND*-gates by the identifiers OC1, OC2, and OC3 where

- OC1 = (Start_W_Fail_S $\wedge$ Wait_W_Fail_S) $\wedge$ ($\neg$failureDeteted $\wedge$ Wait_Mon_Fail),

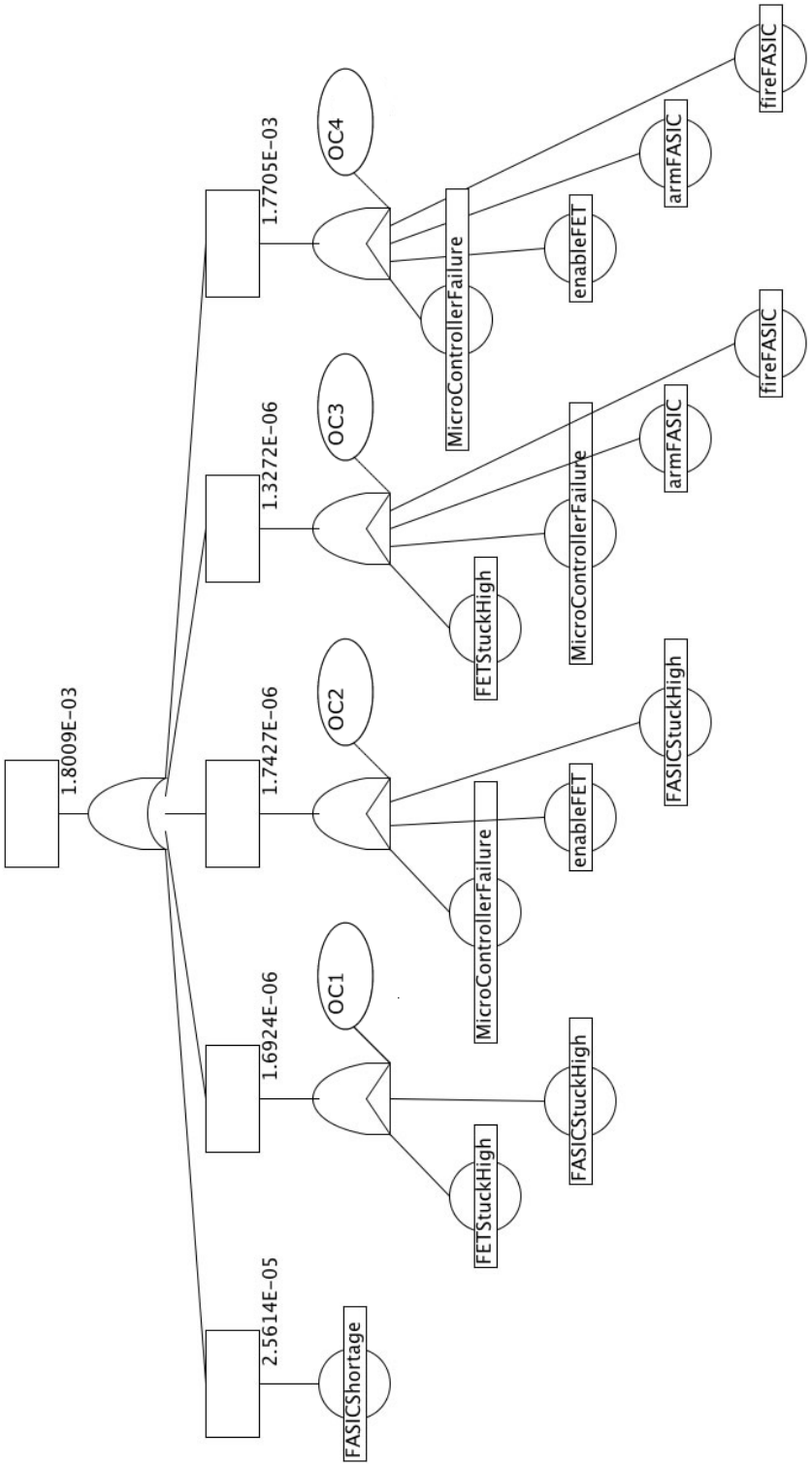


Figure 10.3: Fault tree of the airbag system.

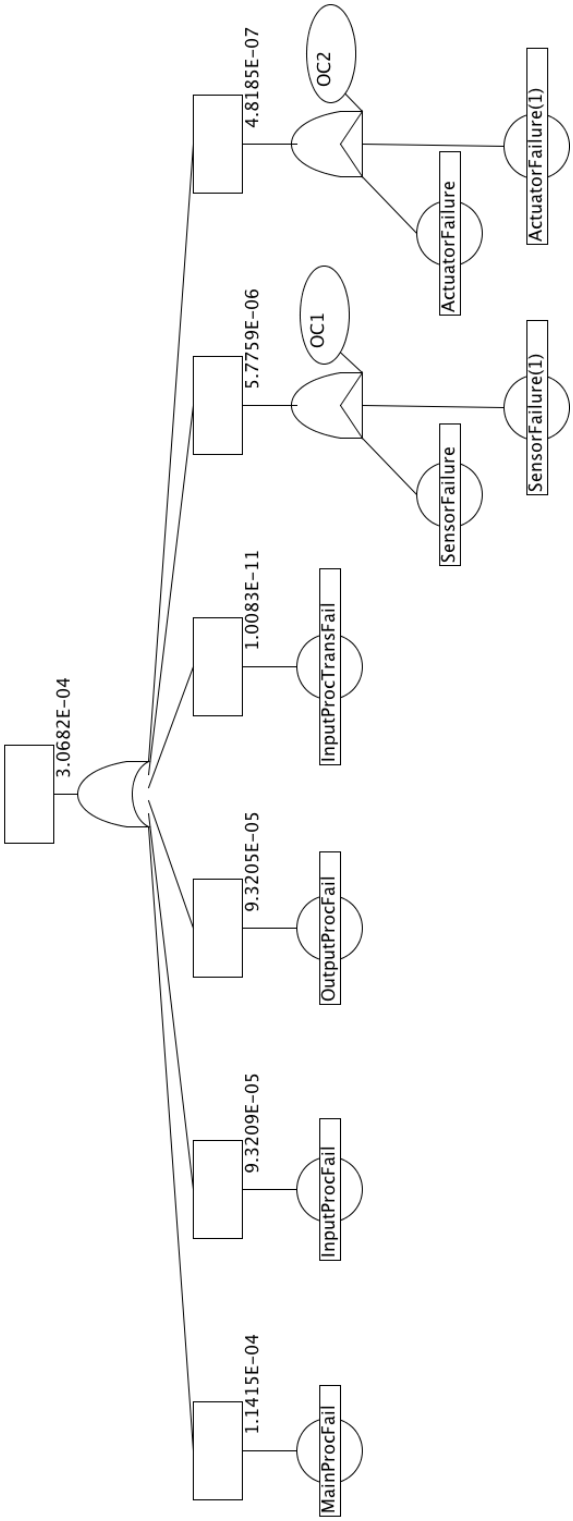


Figure 10.4: Fault tree of the embedded control system.

- OC2 = (Start_W_Fail_F$\triangleleft$Wait_W_Fail_F)$\wedge$(¬failureDeteted$\triangleleft_]$Wait_Mon_Fail) and
- OC3 = Wait_R_Fail $\wedge$ (¬failureDeteted $\triangleleft_]$ Wait_Mon_Fail).

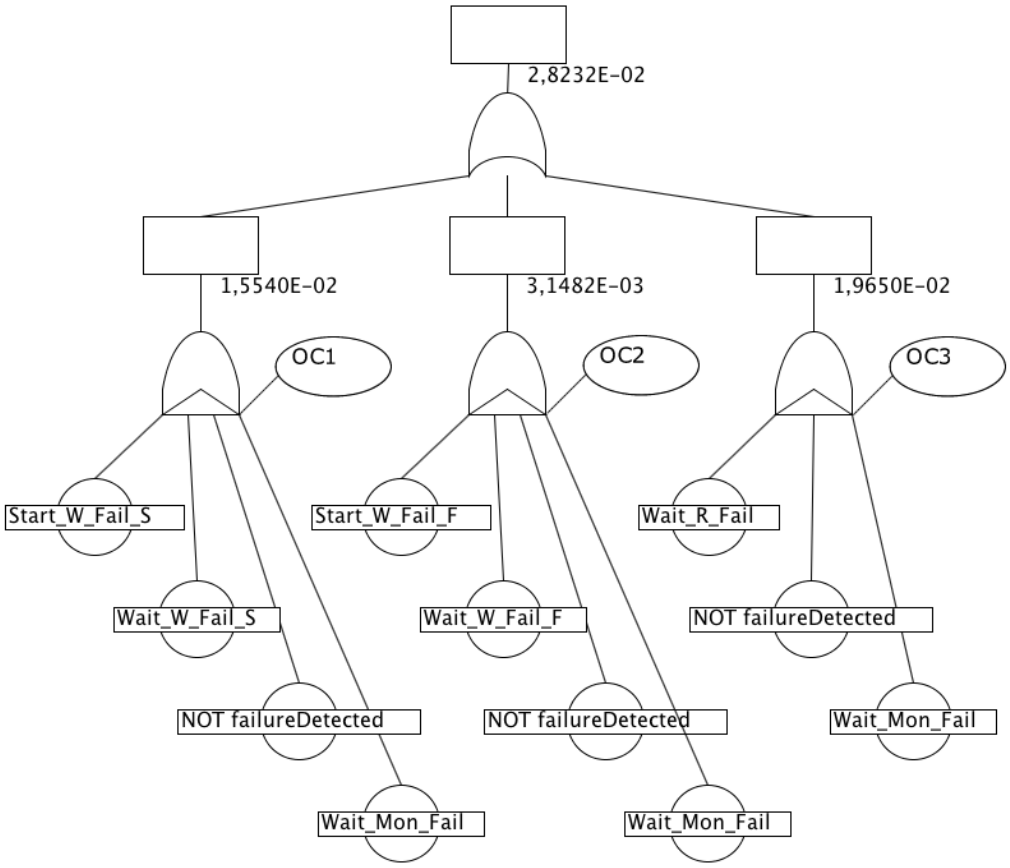


Figure 10.5: Fault tree of the train odometer controller (created with the combined approach).

10.2.5 Airport Surveillance Radar

Figure 10.6 shows the fault tree of the ASR system described in Section 6.6.5. The redundancy in the 2 channel variant of the ASR does not have an impact on the probability of the case where the position information of an aircraft is lost, since the redundant channel is only added in order to increase the availability. Consequently, the fault trees of the 1 channel and 2 channel variant of the ASR are equal. For better readability we have replaced the order constraints of the *PAND*-gates by the identifiers OC1 and OC2, where

- OC1 = (dot_sighted ∧ plot_detected ∧ initTrack ∧ PSR_only ∧ coastTrack) and
- OC2 = (dot_sighted ∧ plot_detected ∧ initTrack ∧ SSR_only ∧ coastTrack).

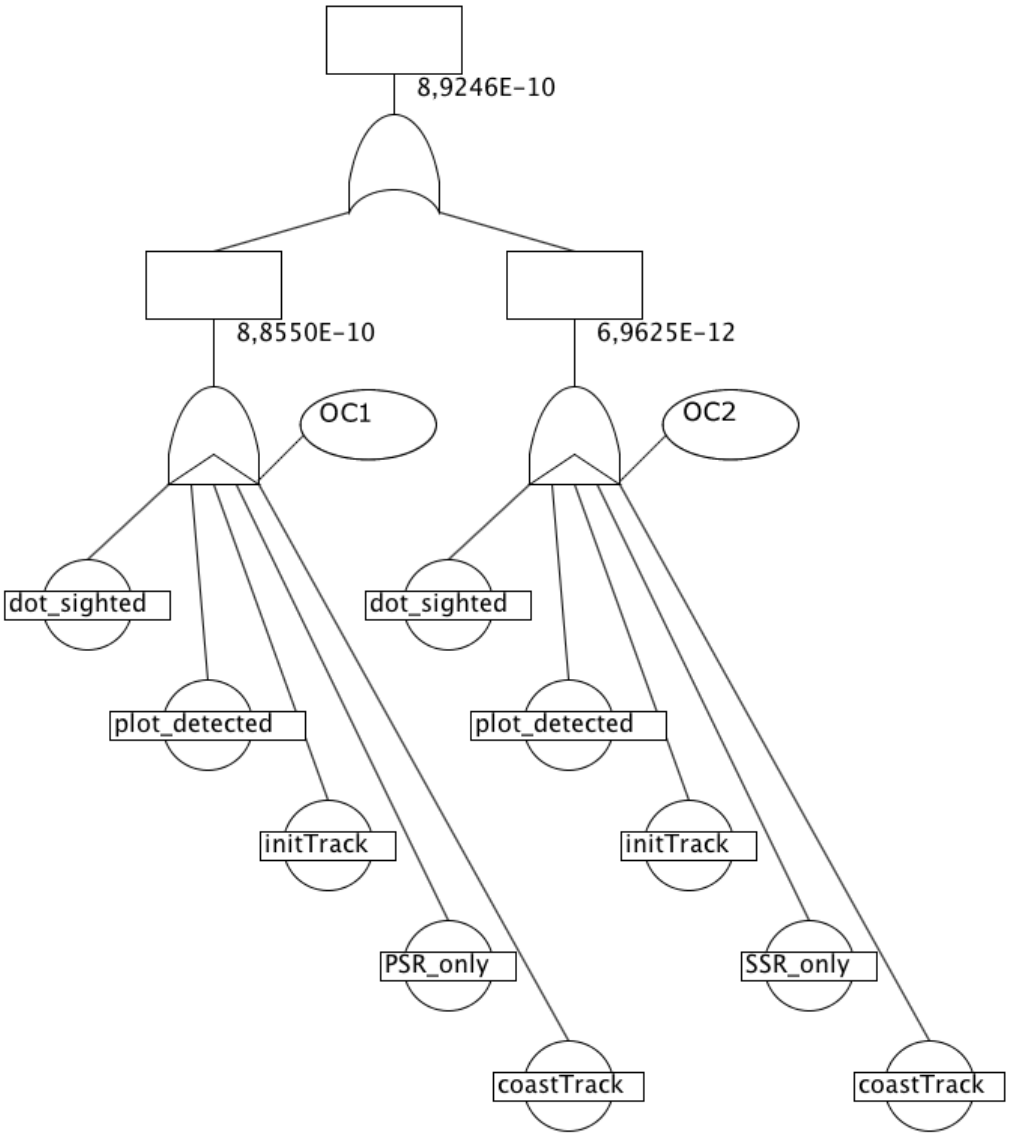


Figure 10.6: Fault tree of the airport surveillance radar system.

10.3 Graphical Representation of Event Orders in Fault Trees

In the fault trees generated by the event order logic to fault tree mapping described in Section 10.2, the causal event orders are captured by adding the event order logic

formula as an order constraint to the PAND-gate representing this causality class. This is necessary since the *PAND*-gate is not expressive enough to express the event orderings that can be expressed with the event order logic. While this mapping gives all information on the causal event orderings to the user, a graphical representation of the causal event orderings is desirable, since a graphical representation might be easier to interpret for the user.

Common graphical representations for partial orderings are Petri nets[89], event structures [102] and directed acyclic graphs. We are interested in finding a graphical representation that is easy to understand for the engineers that have to interpret the representation. This graphical representation shall then be used in order to specify the order constraints of the PAND-gates of the fault tree. We base our graphical representation on directed acyclic graphs. It is not possible to directly use a directed acyclic graph as a graphical representation of EOL formulas, since it is not possible to capture the interval operators $\Bumpeq$, $\Bumpeq_{]}$, $\Bumpeq_{[}$, $\Bumpeq_{<}$, and $\Bumpeq_{>}$ with a directed acyclic graph. We define the following mapping of EOL operators to graphical representations.

Definition 43. *Graphical Representation of Event Order Logic Operators. Let ψ_1 and ψ_2 complex EOL formulas and ϕ a simple EOL formula. The graphical representations of an EOL formula is obtained by recursively applying the following mappings to the parse tree of an EOL formula.*

$\psi_1 \wedge \psi_2$:	$\psi_1 \vee \psi_2$:	$\psi_1 \Bumpeq \psi_2$:	$\phi \Bumpeq_{]} \psi_1$:	$\psi_1 \Bumpeq_{[} \phi$:	$\psi_1 \Bumpeq_{<} \phi \Bumpeq_{>} \psi_2$:
$\underline{\psi_1 \quad \psi_2}$	$\psi_1 \mid \psi_2$	ψ_1	$\Downarrow \phi$	ψ_1	ψ_1
		$\downarrow$	ψ_1	$\Downarrow \phi$	$\Downarrow$
		ψ_2			ϕ
					$\Downarrow$
					ψ_2

If two events are not connected by an arrow there is no order constraint for those two events specified, if they are connected by an arrow the order is read from top to bottom, which means that the event on the top happens before the event on the bottom.

The fault tree of the railroad crossing running example described in Section 3.2 is shown in Figure 10.2. Figure 10.7 show the graphical representation of the EOL formula $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \Bumpeq \mathrm{Cc})) \Bumpeq_{<} \neg\mathrm{Cl} \Bumpeq_{>} \mathrm{Tc}$ representing one of the causality classes of the railroad crossing example.

The graphical representation of the EOL formula can then be used in order to specify the order constraints of the PAND-gate.

10.3.1 Railroad Crossing

Figure 10.7 shows the graphical representation of the causality class represented by the EOL formula $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \Bumpeq \mathrm{Cc})) \Bumpeq_{<} \neg\mathrm{Cl} \Bumpeq_{>} \mathrm{Tc})$ and Figure 10.8 shows the graphical representation of the causality class represented by the EOL formula $(\mathrm{Ta} \wedge (\mathrm{Ca} \Bumpeq \mathrm{Cc})) \Bumpeq_{<} \neg\mathrm{Cl} \Bumpeq_{>} (\mathrm{Gc} \wedge \mathrm{Tc})$.

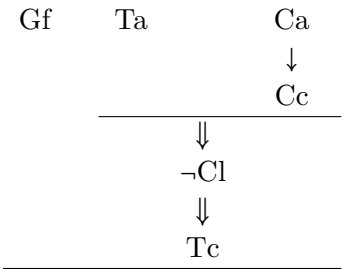


Figure 10.7: Graphical representation of the EOL formula $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\mathbb{\wedge}} \mathrm{Cc})) \mathbin{\mathbb{\wedge}}_{<} \neg\mathrm{Cl} \mathbin{\mathbb{\wedge}}_{>} \mathrm{Tc}$.

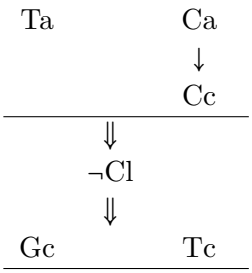


Figure 10.8: Graphical representation of the EOL formula $(\mathrm{Ta} \wedge (\mathrm{Ca} \mathbin{\mathbb{\wedge}} \mathrm{Cc})) \mathbin{\mathbb{\wedge}}_{<} \neg\mathrm{Cl} \mathbin{\mathbb{\wedge}}_{>} (\mathrm{Gc} \wedge \mathrm{Tc})$.

10.3.2 Airbag System

The fault tree for the airbag system described in Section 6.6.2 is shown in Figure 10.3. The order constraints of the *PAND*-gates identified by the identifiers OC1, OC2, OC3 and OC4 can be graphically represented as follows.

- OC1 = (FETStuckHigh $\mathbin{\mathbb{\wedge}}$ FASICStuckHigh) is represented by Figure 10.9,
- OC2 = (MicroControllerFailure $\mathbin{\mathbb{\wedge}}$ enableFET $\mathbin{\mathbb{\wedge}}$ FASICStuckHigh) is represented by Figure 10.10,
- OC3 = (FETStuckHigh $\mathbin{\mathbb{\wedge}}$ MicroControllerFailure $\mathbin{\mathbb{\wedge}}$ armFASIC $\mathbin{\mathbb{\wedge}}$ fireFASIC) is represented by Figure 10.11, and
- OC4 = (MicroControllerFailure $\mathbin{\mathbb{\wedge}}$ enableFET $\mathbin{\mathbb{\wedge}}$ armFASIC $\mathbin{\mathbb{\wedge}}$ fireFASIC) is represented by Figure 10.12.

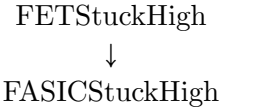


Figure 10.9: Graphical representation of the EOL formula FETStuckHigh $\mathbin{\mathbb{\wedge}}$ FASICStuckHigh.

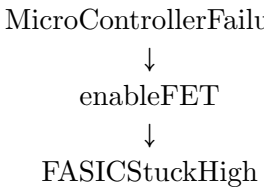


Figure 10.10: Graphical representation of the EOL formula MicroControllerFailure ∧ enableFET ∧ FASICStuckHigh.

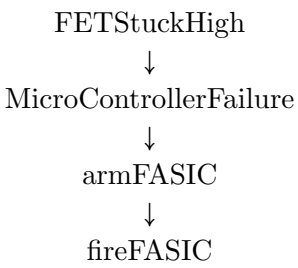


Figure 10.11: Graphical representation of the EOL formula FETStuckHigh ∧ MicroControllerFailure ∧ armFASIC ∧ fireFASIC.

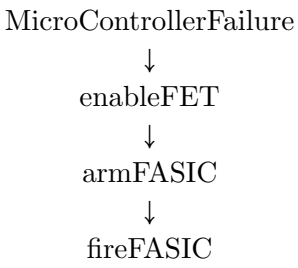


Figure 10.12: Graphical representation of the EOL formula MicroControllerFailure ∧ enableFET ∧ armFASIC ∧ fireFASIC.

10.3.3 Embedded Control System

Figure 10.4 shows the fault tree of the embedded control system presented in Section 6.6.3. The order constraints of the *PAND*-gates OC1 and OC2 are

- OC1 = (SensorFailure ∧ SensorFailure) which is visualized in Figure 10.13 and
- OC2 = (ActuatorFailure ∧ ActuatorFailure) which is visualized in Figure 10.14.

SensorFailure SensorFailure

Figure 10.13: Graphical representation of the EOL formula SensorFailure ∧ SensorFailure.

ActuatorFailure ActuatorFailure

Figure 10.14: Graphical representation of the EOL formula ActuatorFailure ∧ ActuatorFailure.

10.3.4 Train Odometer Controller

The fault tree of the train odometer controller introduced in Section 6.6.4 is shown in Figure 10.5. The order constraints of the *PAND*-gates OC1, OC2, and OC3 are

- OC1 = (Start_W_Fail_S$\,\mathbb{A}\,$Wait_W_Fail_S)∧(¬failureDeteted$\,\mathbb{A}_{]}\,$Wait_Mon_Fail) which is graphically represented in Figure 10.15,
- OC2 = (Start_W_Fail_F$\,\mathbb{A}\,$Wait_W_Fail_F)∧(¬failureDeteted$\,\mathbb{A}_{]}\,$Wait_Mon_Fail) which is graphically represented in Figure 10.16, and
- OC3 = Wait_R_Fail ∧ (¬failureDeteted $\mathbb{A}_{]}$ Wait_Mon_Fail) which is graphically represented in Figure 10.17.

Start_W_Fail_S ⇓ ¬failureDeteted
↓ Wait_Mon_Fail
Wait_W_Fail_S

Figure 10.15: Graphical representation of the EOL formula (Start_W_Fail_S $\mathbb{A}$ Wait_W_Fail_S) ∧ (¬failureDeteted $\mathbb{A}_{]}$ Wait_Mon_Fail).

Start_W_Fail_F ⇓ ¬failureDeteted
↓ Wait_Mon_Fail
Wait_W_Fail_F

Figure 10.16: Graphical representation of the EOL formula (Start_W_Fail_F $\mathbb{A}$ Wait_W_Fail_F) ∧ (¬failureDeteted $\mathbb{A}_{]}$ Wait_Mon_Fail).

Start_R_Fail ⇓ ¬failureDeteted
Wait_Mon_Fail

Figure 10.17: Graphical representation of the EOL formula Wait_R_Fail ∧ (¬failureDeteted $\mathbb{A}_{]}$ Wait_Mon_Fail).

10.3.5 Airport Surveillance Radar

Figure 10.6 shows the fault tree of the ASR system described in Section 6.6.5. The order constraints of the *PAND*-gates OC1 and OC2 are

- OC1 = (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ PSR_only $\wedge$ coastTrack) which is represented by Figure 10.18 and
- OC2 = (dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ SSR_only $\wedge$ coastTrack) which is represented by Figure 10.19.

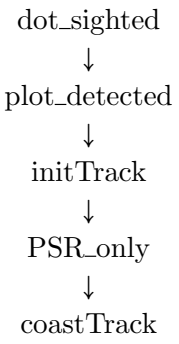


Figure 10.18: Graphical representation of the EOL formula dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ PSR_only $\wedge$ coastTrack.

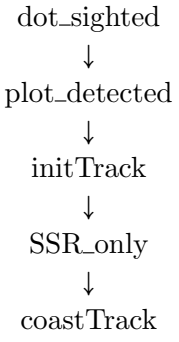


Figure 10.19: Graphical representation of the EOL formula dot_sighted $\wedge$ plot_detected $\wedge$ initTrack $\wedge$ SSR_only $\wedge$ coastTrack.

10.4 Relationship to Minimal Cut Set Analysis

In fault tree analysis a minimal cut set (MCS) [97] is defined as the minimal boolean combination of basic events that can cause the top level event. This means that no true subset of the events in a minimal cut set can cause the top level event. Each of the causality classes returned by the causality checker corresponds to a minimal cut

set, since it represents a minimal boolean combination of basic events that cause the property violation represented by the top level event.

Theorem 64. *Each causality class corresponds to a minimal cut set causing the property violation.*

Proof. For a causality class not to be a minimal cut set of a property violation there would have to exist a subset of the events that also causes the property violation. Due to the minimality constraint that is imposed by the actual causality conditions, no such subset exists for a causality class. Consequently, each causality class that is returned by the causality checker corresponds to a minimal cut set of the property violation. □

10.5 Root-Cause Identification

Causality checking identifies the causal event combinations for an effect or hazard. In the safety analysis of systems one is also interested in identifying the root-cause for a hazard. According to Ladkin [68] a *root-cause* is a necessary causal factor which has no causal predecessor. Intuitively, a root-cause is an event that is considered to be responsible for the hazard to occur. The other events that are being considered as causes, but which are not root-causes, are either necessary in order to enable the root-cause event or are necessary to propagate the failure caused by the root-cause through the system until a hazard occurs. The events train approaching and car approaching in the railroad example, for instance, are events that are identified as being causal by the causality checker, since they are essential for both the car and the train being on the crossing at the same time, but are not considered to be root-causes, since they also happen on good traces. It is not possible for the causality checking approach to distinguish between events that represent root-causes and events that enable the root-causes or mediate between the root-causes and the hazard. The reason for this is that if an event enabling the root-cause or mediating between the root-cause and the hazard is removed from a trace, then the property is not violated any more and the trace does not satisfy the AC1-AC3 conditions.

The identification of root-causes is left for future research, nevertheless we propose a potential approach that allows for the identification of root-causes. It is not possible to automatically discriminate between root-cause events and other causal events. Consequently, we envision a interactive approach where the user indicates events that she suspects to be root-causes or suspects to be non-root-cause events. The approach shall be implemented as an interactive user guided refinement step of the causality checking result. For the causality class specified by the EOL formula $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_{<} \neg\mathrm{Cl} \barwedge_{>} \mathrm{Tc}$ of the railroad crossing example the suspected root-causes could, for instance, be the events Gf and ¬Cl. For these events it is then checked whether there exists a causal predecessor. This is achieved by computing the causal events for the suspected root-cause. The computation of the causal predecessor is limited on the set of traces that belong to the causality class specified by

the EOL formula. If no causal predecessor for a suspected root-cause can be found, it is indeed a root-cause. If a causal predecessor is found, this causal predecessor is treated as a suspected root-cause and the test is repeated. In our example both the event Gf and the event ¬Cl are identified as root-causes. Obviously, the result of the proposed root-cause computation relies heavily on the subjective identification of potential root-causes by the user and it needs to be investigated in future research whether the proposed approach provides valuable results.

Chapter 11

Industrial Application Scenarios

The content of this chapter is based on the publications [73].

Contents

11.1 Introduction

Tool support and a seamless integration into the industrial development processes and the industrial tool landscape are key factors for the success of a method in an industrial setting. For causality checking to be applicable a PRISM or Promela analysis model of the system to be analyzed is necessary. While it is possible for engineers to construct the analysis models manually, this is a time consuming task. We integrate causality checking into the QuantUM framework [72], a method allowing for the generation of analysis models from higher-level architectural description languages such as the unified modeling language (UML) [86] or the systems modeling language (SysML) [52]. In Section 11.2 we discuss how causality checking can be integrated into the QuantUM tool and in Section 11.3 we give examples how the resulting model-based safety analysis method can be used in an industrial engineering process and within the context of the ISO 26262 [55] standard for functional safety in the automotive domain.

11.2 Embedding Causality Checking in QuantUM

The QuantUM tool automatically generates a PRISM model and corresponding properties from the annotated UML or SysML model. The PRISM model and properties are taken as an input for the causality checking. The causality checker then performs the causality analysis and the results are represented by a fault tree visualization using the event order logic to fault tree mapping proposed in Chapter 10. Individual traces represented by the causality classes can be displayed in UML by UML sequence diagrams.

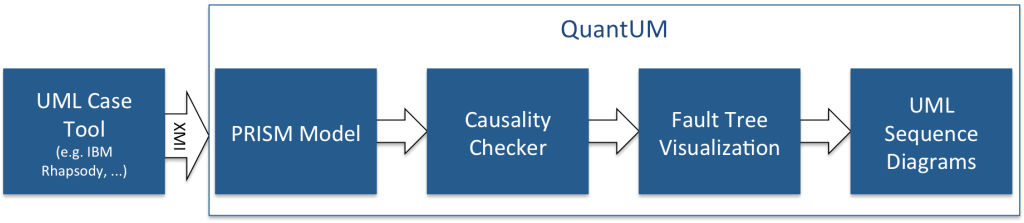


Figure 11.1: Causality Checking integration into the QuantUM tool chain.

Figure 11.1 shows the QuantUM toolchain and the integration of the causality checker. The causality checker is encapsulated by the QuantUM tool. This means that the engineers using QuantUM will not need to have an in-depth understanding of causality checking or the formal methods involved.

11.3 Causality Checking in Model-Based Safety Analysis

In this section we will discuss how the causality checking method integrated into the QuantUM approach can be used in an engineering process based on the V-model [26]. Figure 11.2 depicts the different stages of the V-model. The stages where QuantUM and causality checking can be applied are colored in dark blue and marked with the QuantUM logo.

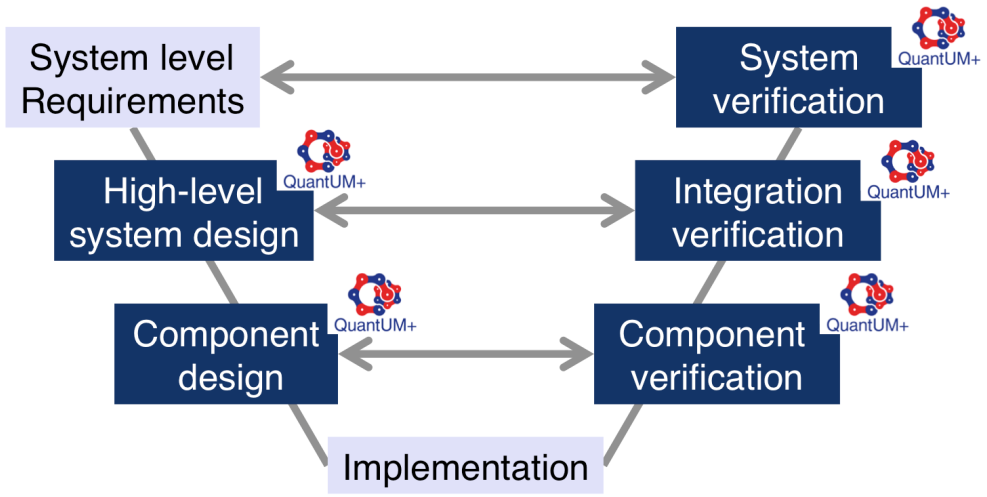


Figure 11.2: V-model for systems engineering.

At all stages of the V-model QuantUM can be used to perform fast checks whether certain unsafe system sates are reachable and what the probability of reaching such an unsafe system state is. These checks allow to demonstrate the

effectiveness of introduced fault tolerance and safety mechanisms. In case an unsafe system state is reachable a fault tree documenting the causal events for reaching the unsafe state will be generated and presented to the user.

Furthermore, QuantUM can be used to do a fast evaluation of different architecture alternatives during the high-level system design stage and the component design stage. The probabilities for the property violations computed by QuantUM and the generated fault trees can be used in order to compare the safety of different architectural variants. When considering, for instance, the airbag system introduced in Section 6.6.2, QuantUM can answer the question whether an extended variant with two redundant crash evaluation paths is safer than the described variant with only one crash evaluation path. Additionally, it is possible to determine whether the second decision path can be bypassed by design faults. Note that the above described use cases were previously preformed manually by highly trained domain experts and can now be automatically executed by the QuantUM tool with the integrated causality checker.

During the verification stages the effectiveness of the implemented fault tolerance and safety mechanisms can be analyzed, by computing the fault trees for safety-critical property violations. In addition, it is possible to compare the fault trees that were automatically generated by QuantUM with fault trees that where manually created by engineers. This allows to check whether the engineer made correct assumptions when creating the manual fault tree or whether there is a deviation from the manual fault tree and the behavior of the system model.

11.3.1 QuantUM and Causality Checking in the Context of the ISO 26262

The airbag case study described in Section 6.6.2 falls within the scope of the ISO 26262 [55] standard that defines processes and techniques in support of a safe design and implementation of passenger automobile systems containing safety relevant or safety-critical programmable electronic components. In this section we will discuss how QuantUM and causality checking can be used in order to support the activities recommended by the ISO 26262 standard. The ISO 26262 can be viewed as an application domain specific specialization of the IEC 61508 [54] standard applicable to general programmable, safety-critical electronic systems. The ISO 26262 is considered to represent the state of the art with respect to functional safety processes and techniques in the automotive system engineering area, and is hence likely to become a point of reference for litigation in this domain. In the ISO 26262 a process model as well as different methods and work products are defined. The standard is structured into nine parts that address different stages of the system design process.

- Part 26262-1 presents a definition of terms used in the standard.
- Part 26262-2 defines requirements on the management of functional safety. The standard defines different confirmation measures to be performed including confirmation reviews, functional safety audits and functional safety

assessments. These process steps are designed to compile a safety case. A safety case is defined to be an argument that the safety goals for an *item* are satisfied based on the functional safety assurance measures carried out. In the ISO 26262 an item denotes a system, a collection of systems or a function to which ISO 26262 is applied. In this part a life-cycle model is proposed which includes all necessary processes that have to be applied to the new system.

- Part 26262-3 is devoted to the concept phase of the system development life cycle. This stage is pivotal for the functional safety of the system since it involves the planning of all safety assurance activities. In particular, it comprises the definition of the different system aspects that are safety relevant, the initiation of the safety process, a hazard analysis and a risk assessment. In this part a qualitative method is defined, which helps to determine the Automotive Safety Integrity Level (ASIL) of a system aspect. The different ASIL levels range from QM (not safety-critical), A (lowest safety-criticality) to D (highest safety-criticality). The goals of the concept phase include the derivation of a functional safety concept and to derive functional safety goals.

- Part 26262-4 is concerned with the system design and with ensuring that the system design satisfies the technical safety requirements defined in the concept phase. The use of deductive analysis techniques, such as Fault Tree Analysis (FTA), is proposed in this part and they are highly recommended for ASIL level C and D. Inductive analysis techniques, such as Failure Mode and Effect Analysis (FMEA) [53], Event Tree Analysis (ETA) and Markov modeling are highly recommended for all ASIL levels. For every aspect which is integrated into the new system an integration and testing strategy has to be defined based on the system design specification.

- Part 26262-5 is devoted to hardware design and to ensuring the compliance with the former defined safety goals of the system. The analysis techniques to verify the hardware components are also FTA, FMEA and others depending on the ASIL levels. An important part of the hardware verification is the evaluation of the probability of safety goal violations due to random hardware failures. Depending on the assigned ASIL level concrete probability limits for safety goal violations are specified by the ISO 26262 standard.

- Part 26262-6 defines the software development process including the definition of software safety requirements and their verification. In this part the V-model is proposed as the applicable software process and life-cycle model. Software safety requirements are derived from the technical safety concept and the system design. Further objectives of this stage are a refinement of the software-hardware interface, and an assurance mechanism showing that the software safety requirements are consistent with the technical safety concept and the system design specification.

- Part 26262-7 addresses safety relevant issues in the production, operation and

maintenance of automotive systems and is not directly relevant in the context of this thesis.

- Part 26262-8 describes supporting processes like requirements management, configuration management, verification, documentation and the qualification of software tools, software and hardware.

- Part 26262-9 addresses ASIL oriented and safety-oriented analysis techniques. Qualitative FMEA as well as qualitative FTA and Event Tree Analysis (ETA) are listed as qualitative analysis techniques. As quantitative analysis techniques, quantitative FMEA, quantitative FTA, ETA and Markov models are recommended, amongst others.

We will now discuss how QuantUM and causality checking can be used in order to support the different activities recommended by the different parts of the ISO 26262 standard.

- Regarding part 26262-1 it is most interesting how terms related to formal methods are defined. A *formal notation* is defined as a description technique that possesses completely defined syntax and semantics. A notation is defined to be *semi-formal* if only its syntax is precisely defined. The UML and SysML would hence qualify as a semi-formal notation. *Formal verification* is defined as a method that allows one to prove correctness of a system against its specification. A verification method is *semi-formal* if it is based on specification given in a semi-formal notation. Following this terminology, the QuantUM approach uses both semi-formal notations, like UML and the QuantUM extension, as well as a formal notation using the Promela and PRISM language. Causality checking as well as model checking classify as formal verification methods according to the ISO 26262.

- Amongst others, part 26262-2 requires confirmation reviews using Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) to be performed. For ASIL A and ASIL B this is merely required, for ASIL C this analysis shall be performed by a person not belonging to the development team, and for ASIL D by a person not belonging to the same organization or department within an organization. QuantUM allows for an automated FMEA and with the causality checking integration for an automated FTA.

- The main activities described in part 26262-3 are the item definition, the hazard analysis and risk assessment, and the specification of the functional safety concept.

  - In the item definition all relevant aspects of the system are defined. A description of the functionality of the item has to be given. The dependencies and interaction to other items and the environment are specified in order to give a contextual view of the item. The description of the

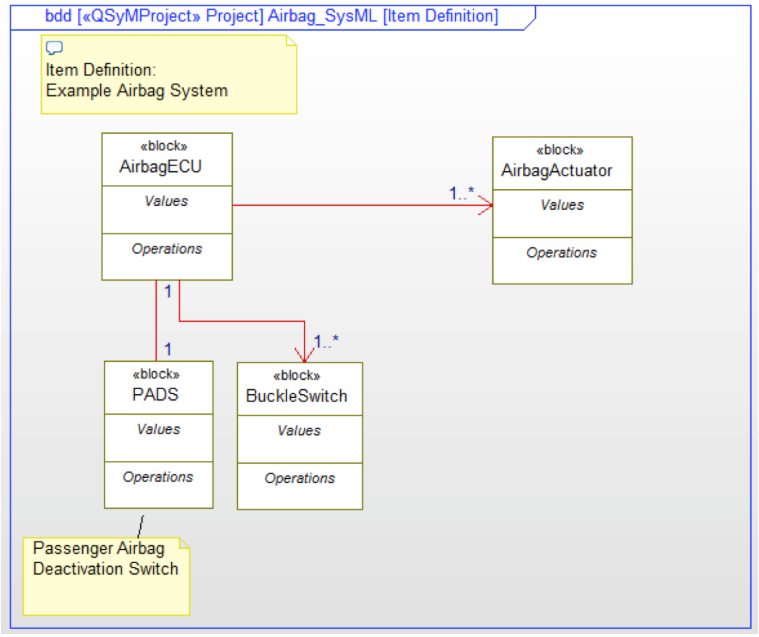


Figure 11.3: A contextual view of the airbag control unit.

item supports an adequate understanding of the item so that the activities in subsequent phases can be performed. We propose that a high level UML / SysML diagram is generated by the user which relates the item with its environment and other items in order to give an overview. A coarse description of the item and its environment is given using a high level UML Statechart for the different items in the context. In Figure 11.3 an exemplary context view of the airbag system, described in detail in Section 6.6.2, is given relating the airbag electrical control unit (AirbagECU) to its neighboring components, for instance the passenger airbag deactivation switch.

– In the hazard analysis and risk assessment (HARA) the hazards of the item are identified. A categorization of the malfunctions that can trigger the hazards has to be given and safety goals related to the prevention or mitigation of the hazardous events have to be formulated. Given a use case diagram depicting the system functionality, hazardous malfunctions can be identified. An example of such a use case diagram showing the malfunctions that can be derived from the use case “deploy airbag in crash situation” is shown in Figure 11.4.

– In the functional safety concept phase safety requirements have to be derived from the former defined safety goals. The requirements are allocated to the preliminary architectural elements, items or external objects. We propose that the preliminary architecture is modeled in UML / SysML using structural and behavioral diagrams to describe the

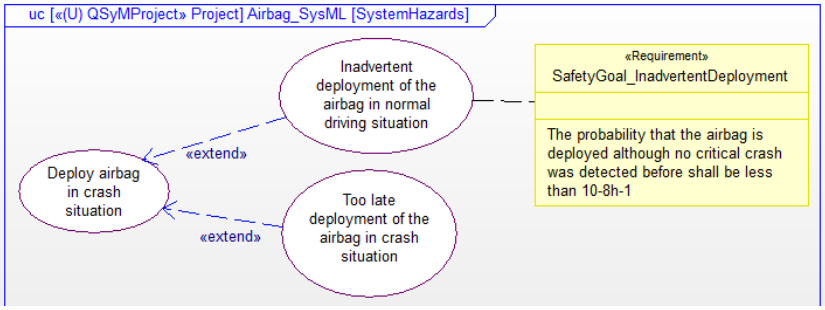


Figure 11.4: Safety goals generated in the Hazard Analysis and Risk Assessment.

static and dynamic aspects of the system. The requirements can then be directly allocated to architectural elements in UML or SysML. The QuantUM approach and causality checking can then be used in order to analyze the preliminary architecture. The preliminary architectural elements of the airbag system are shown in Figure 11.5.

- Part 26262-4 is concerned with the *system design* and with ensuring that the system design satisfies the safety requirements. In this step a refinement of the functional safety concept is performed. In order to do so the preliminary architecture model created in the functional safety concept phase is refined. The requirements gained from this step are again allocated to the UML / SysML elements as done in the previous step. In order to demonstrate the effectiveness of the proposed fault tolerance and safety mechanisms, that are introduced into the model in this phase, a proof of concept analysis using the QuantUM tool can be conducted. The use of deductive analysis techniques, such as FTA, are proposed and highly recommended for the ASIL levels C and D. Inductive Analysis techniques, such as FMEA, Event Tree Analysis (ETA) and Markov modeling are endorsed for all ASIL levels. The causality checking integration in QuantUM allows for the automatic generation of fault trees from design models, furthermore QuantUM provides support for carrying out a probabilistic FMEA.

- Part 26262-5 is devoted to *hardware design*. While we note that hardware analysis can be carried out by model checking [8], the QuantUM approach is not really suitable to describe pure hardware architectures. With the QuantUM approach it is, nonetheless, possible to analyze hardware on a system level. Part 26262-5 requires an evaluation of safety goal violations caused by random hardware failures. Using the QuantUM UML/SysML profile it is possible to attach failure rates and probabilities directly to component failures in the UML / SysML model. The QuantUM tool can then be used to automatically check whether safety goals are violated by random hardware failures and what the probability of such a violation is. In addition the causes of a safety goal violation are computed by the causality checker integrated in the

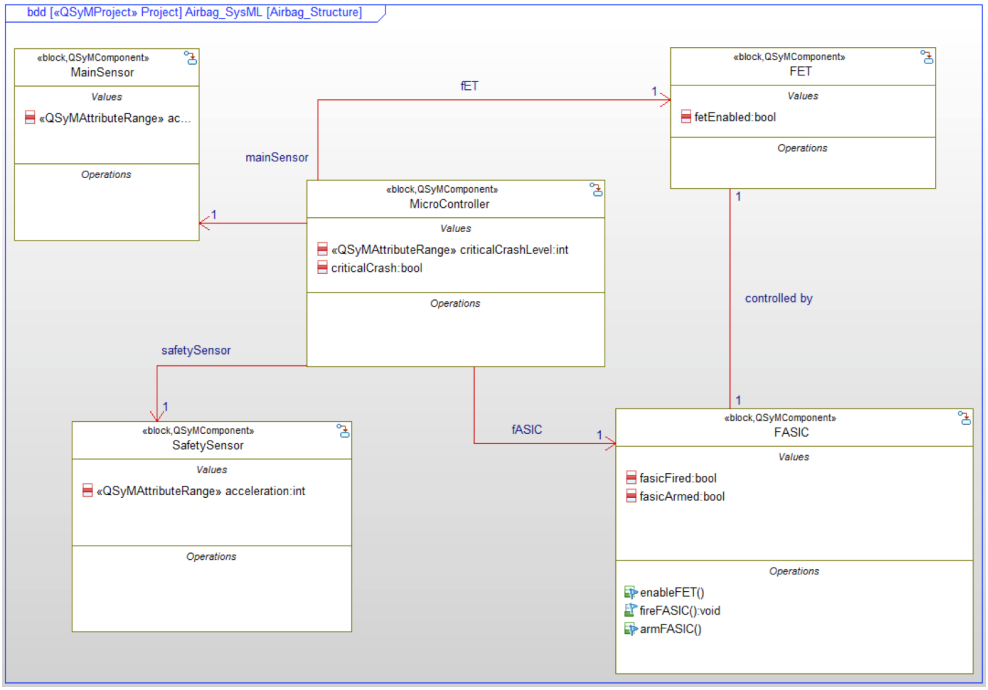


Figure 11.5: The structural overview diagram of the functional safety concept.

QuantUM tool and together with their corresponding probability depicted in a fault tree. The fault tree of the airbag example is depicted in Figure 11.6.

- In part 26262-6 a formal verification for ASIL levels C and D is recommended. The use of formal notations for software unit design is recommended for all ASIL levels. The QuantUM approach uses a semi-formal notation and synthesizes a formal model of individual subsystems, which fits the former two requirements. Semantic code analysis is one of the methods that are specified in this part of the standard and is recommended for all ASIL levels. Such an analysis can be accomplished by model checking, and, consequently also by causality checking. In [15] the consistency of the translation semantics from high level UML / SysML models to the model checking code QuantUM is generating with code generation semantics for a widely used industrial CASE tool is shown.

- Part 26262-7 addresses safety relevant issues in the production, operation and maintenance and is not directly relevant in the context of this thesis.

- In Part 26262-8 supporting processes are discussed and hence there is no direct application scenario for QuantUM or causality checking within this part.

- A major concern of Part 26262-9 is the tailoring of the Automotive Safety Integrity Level (ASIL) during the refinement of the system architecture which is described as follows. The ASIL of the safety goals of an item under devel-

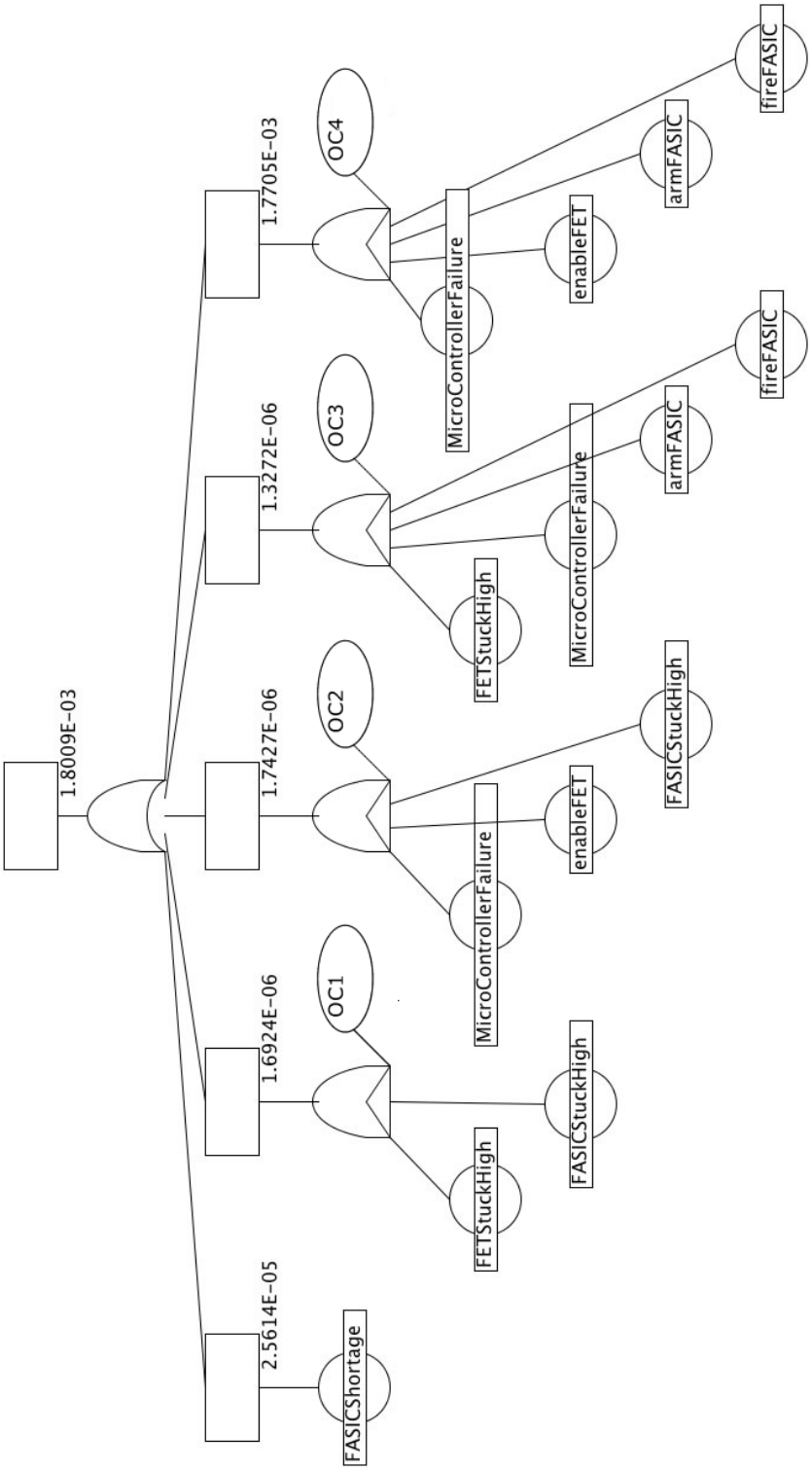


Figure 11.6: Fault tree for an inadvertent deployment of the airbag example.

opment is propagated throughout the item's development process. Starting from safety goals, the safety requirements are derived and refined during the development phases. The ASIL, as an attribute of the safety goal, is inherited by each subsequent safety requirement. The functional and technical safety requirements are allocated to architectural elements, starting with preliminary architectural assumptions and ending with the hardware and software elements. The method of ASIL tailoring during the design process is called "ASIL decomposition". If an architectural element is derived into two architectural elements that are sufficiently independent with respect to the violation of safety goals the ASIL can be decomposed on the lower level. This offers the opportunity

1. to implement safety requirements redundantly by independent architectural elements, and
2. to assign a potentially lower ASIL to these decomposed safety requirements.

If the architectural elements are not sufficiently independent, then the redundant requirements and the architectural elements inherit the initial ASIL. This means that if it is possible to prove a sufficient independence in the functionality of redundant components a possibly lower ASIL can be attached to the component and, thus, the cost of developing the redundant components is decreased. The ISO 26262 standard imposes the following requirements on the independence of architectural elements:

- The elements are sufficiently independent if the analysis of dependent failures does not find a cause of dependent failures that can lead to the violation of a safety requirement before decomposition,
- or if each identified cause of dependent failures is controlled by an adequate safety measure according to the ASIL of the safety goal.

In other words, if both architectural elements have to fail independently from each other to cause a violation of the top-level safety requirement they are sufficiently independent.

In terms of causality relationships this means that:

- Both architectural elements have to fail in order to violate the top-level safety requirement. This is the case if they are connected by an AND-gate in the fault tree that is generated for the violation of the safety requirement.
- There is no relative causal ordering between the failures of the two architectural elements.

The above mentioned causality relationships can be computed by the causality checker integrated in the QuantUM tool, thus it is possible to automatically

detect whether two architectural elements are sufficiently independent with respect to the violation of safety goals and hence ASIL decomposition can be applied. The fault tree returned by the causality checker for the airbag system, for instance, clearly indicates that no ASIL decomposition for the components FASIC, FET and microcontroller is possible.

11.3.2 Summary

We have shown that QuantUM and causality checking can be used in order to support the activities in an engineering process. Some of the activities that previously were executed manually, like fault tree analysis, can be automated using the causality checking approach. Consequently, both the QuantUM approach and causality checking provide the potential to reduce the development costs of safety-critical systems.

Chapter 12

Conclusion

Conclusion

In this thesis we have proposed the causality checking method for automated causality reasoning in system models. The method can be used to compute the causal event combinations for the violation of a non-reachability property in a system model.

We introduced the event order logic, which is a temporal logic that allows to formally capture the occurrence and order of events and is used to represent the results of the causality checking method.

We have shown how causal relationships can be inferred in system models based on an adapted version of the actual cause definition by Halpern and Pearl and how the order of the events can be taken into account as a causal factor. We proposed an on-the-fly algorithm for causality checking and showed how it can be integrated into the state-space exploration algorithms used in qualitative model checking. Furthermore, we extended the causality checking method in order to be applicable to probabilistic system models. Since a pure probabilistic causality checking method entails a high performance penalty for the necessary probability computation, as became clear in this thesis, we showed how this bottleneck can be mitigated by a combination of qualitative and probabilistic causality checking. We demonstrated the applicability and usefulness of causality checking on several case studies from industry and academia.

The proposed causality checking method enables an automated causal analysis of concurrent systems including hardware and software. The results generated by the causality checker provide valuable insight as to why the hazard or property violation occurred, which is very tedious or even impossible to determine if standard model checking and manual counterexample analysis is used, due to the amount of counterexamples generated. The adapted causality notion that we use and more specifically, considering the event ordering as a causal factor, makes causality checking more suitable for the analysis of concurrent systems than the existing causal reasoning approaches based on the counterfactual argument or the original Halpern and Pearl actual cause definition. Furthermore, causality checking, in contrast to the existing causality reasoning approaches, focuses on identifying all possible causal event combinations instead of focusing on the identification of a single causal event in a single counterexample trace.

Even though the causality checking method scaled well for the case studies

presented in this thesis, we discussed how the scalability of causality checking can be further increased.

In order to make the results of causality checking easy to interpret for engineers, we proposed a mapping of event order logic formulas to fault trees, a method used in industry to reason about the relationships between a property violation and the corresponding causal events. In addition, we discussed how causality checking can be integrated into the QuantUM method, a framework for the automated safety analysis of system and software architectures and gave example application scenarios for the QuantUM method including causality checking in an industrial engineering process.

The mapping of the causality checking results to fault trees and the integration of the causality checking approach into the QuantUM framework facilitates the usage of causality checking in an industrial safety-engineering process. With the combination of QuantUM and causality checking, some of the activities that previously had to be executed manually, like fault tree analysis, can be automated. Consequently, both the QuantUM approach and causality checking provide the potential to reduce the development costs of safety-critical systems.

Future Work

For the time being the causality checking is limited to non-reachability properties. In future work the causality reasoning could be extended in order to also support causality checking for liveness properties. In contrast to non-reachability properties where a counterexample consists of a finite trace, the counterexample for a liveness property consists of an infinite trace. The extension would require to define the notion of causality for the violation of a liveness property on an infinite trace and furthermore the extension of the proposed algorithms.

In order to further increase the scalability of the qualitative causality checking approach, in future work it could be investigated whether the information obtained by the static analysis performed for partial order reduction can be used to identify all causal event orders even if partial order reduction is enabled during the state-space exploration. The key idea is to store the information about the event orderings that have been pruned by the partial order reduction, instead of enumerating all possible event orderings through state-space exploration.

The root-cause computation proposed in Chapter 10 needs to be implemented as an interactive user guided refinement step of the causality checking results and it needs to be evaluated whether the approach provides valuable results.

List of Figures

List of Tables

Listings

Bibliography

[1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying Continuous-Time Markov Chains. In *Proceedings of the Eighth International Conference on Computer Aided Verification (CAV 1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 269–276. Springer, 1996. 19

[2] 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 *Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems (QEST 2009)*, pages 209–308. IEEE Computer Society, 2009. 3, 93

[3] H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov. DiPro - A Tool for Probabilistic Counterexample Generation. In *Proceedings of the 18th International SPIN Workshop (SPIN 2011)*, volume 6823 of *Lecture Notes in Computer Science*, pages 183–187. Springer, 2011. 108, 118

[4] H. Aljazzar and S. Leue. Debugging of Dependability Models Using Interactive Visualization of Counterexamples. In *Proceedings of the Fifth International Conference on Quantitative Evaluation of Systems (QEST 2008)*, pages 189–198. IEEE, 2008. 10

[5] H. Aljazzar and S. Leue. Directed Explicit State-Space Search in the Generation of Counterexamples for Stochastic Model Checking. *IEEE Transactions on Software Engineering*, 36(1):37–60, 2009. 3, 10, 19, 20, 107

[6] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-Checking Continuous-Time Markov Chains. *ACM Transactions on Computational Logic (TOCL)*, 1(1):162–170, 2000. 19

[7] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003. 18, 19, 134

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. 1, 3, 14, 16, 20, 59, 90, 116, 132, 139, 144, 173

[9] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, volume 38, pages 97–105. ACM, 2003. 7

[10] J. Barnat, L. Brim, and P. Ročkai. Scalable multi-core LTL model-checking. In *Proceedings of the 14th International SPIN Workshop (SPIN 2007)*, volume 4595 of *Lecture Notes in Computer Science*, pages 187–203. Springer, 2007. 88

[11] J. Barnat and P. Ročkai. Shared Hash Tables in Parallel Model Checking. *Electronic Notes in Theoretical Computer Science*, 198(1):79–91, 2008. 88

[12] A. Beer, U. Kuehne, F. Leitner-Fischer, and S. Leue. Towards Symbolic Causality Checking using SAT-Solving. In *Proceedings of Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2014). Dagstuhl, Germany*, 2014. 9, 10

[13] A. Beer, U. Kühne, F. Leitner-Fischer, S. Leue, and R. Prem. Analysis of an Airport Surveillance Radar using the QuantUM approach. Technical Report soft-12-01, Chair for Software Engineering, University of Konstanz, Germany, 2012. Available from: `http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-12-01.pdf`. 5, 67, 98

[14] A. Beer, U. Kühne, F. Leitner-Fischer, S. Leue, and R. Prem. Model-based Formal Safety Analysis of an Airport Surveillance Radar System (short paper). In *Participant Proceedings of 12th International Workshop on Automated Verification of Critical Systems (AVOCS 2012). Bamberg, Germany*, 2012. 5, 67

[15] A. Beer, U. Kühne, F. Leitner-Fischer, S. Leue, and R. Prem. Quantitative Safety Analysis of Non-Deterministic System Architectures. Technical Report soft-13-02, Chair for Software Engineering, University of Konstanz, Germany, 2013. 174

[16] A. Beer, F. Leitner-Fischer, and S. Leue. On the Relationship of Event Order Logic and Linear Temporal Logic. Technical Report soft-14-01, Chair for Software Engineering, University of Konstanz, Germany, 2014. Available from: `http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-14-01.pdf`. 5, 29

[17] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining Counterexamples Using Causality. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2009. 8, 9, 10

[18] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining Counterexamples Using Causality. *Formal Methods in System Design*, 40(1):20–40, 2012. 8, 9, 10

[19] A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002. 140, 141

[20] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*,

volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. 10, 140, 141

[21] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels. Paths to Property Violation: a Structural Approach for Analyzing Counter-Examples. In *Proceedings of the IEEE 12th International Symposium on High-Assurance Systems Engineering (HASE 2010)*, pages 74–83. IEEE, 2010. 9

[22] E. Böde, T. Peikenkamp, J. Rakow, and S. Wischmeyer. Model Based Importance Analysis for Minimal Cut Sets. In *Proceedings of the Sixth International Symposium on Automated Technology for Verification and Anaylsis (ATVA 2008)*, volume 5311 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2008. 10, 96

[23] H. Boudali, P. Crouzen, and M. Stoelinga. A Rigorous, Compositional, and Extensible Framework for Dynamic Fault Tree Analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(2):128–143, 2010. 10

[24] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *Proceedings of the Fifth International Symposium on Automated Technology for Verification and Anaylsis (ATVA 2007)*, volume 4762 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2007. 9

[25] M. Bozzano and A. Villafiorita. Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform. In *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2003)*, volume 2788 of *Lecture Notes in Computer Science*, pages 49–62. Springer, 2003. 9

[26] A.-P. Bröhl. *Das V-Modell*. Oldenbourg, 1995. 168

[27] CENELEC. CENELEC EN 50128:2011 Railway Applications - Communication, Signalling and Processing Systems - Software for Railway Control and Protection Systems, 2011. 1

[28] A. K. Chandra and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981. 23, 24

[29] H. Chockler, J. Y. Halpern, and O. Kupferman. What Causes a System to Satisfy a Specification? *ACM Transactions on Computational Logic (TOCL)*, 9(3):20:1–20:26, 2008. 9

[30] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. 19

[31] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking (3rd ed.)*. The MIT Press, 2001. 1, 7, 14

[32] J. Collins, editor. *Causation and Counterfactuals*. MIT Press, 2004. 2, 8, 55, 56

[33] M. de Jonge and T. Ruys. The SpinJa Model Checker. In *Proceedings of the 17th International SPIN Workshop (SPIN 2010)*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer, 2010. 14, 88, 92, 145

[34] H. Debbi and M. Bourahla. Causal Analysis of Probabilistic Counterexamples. In *Proceedings of the 11th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013)*, pages 77–86. IEEE, 2013. 10

[35] H. Debbi and M. Bourahla. Generating Diagnoses for Probabilistic Model Checking Using Causality. *Journal of Computing and Information Technology*, 21(1):13–23, 2013. 10

[36] L. Dillon, G. Kutty, L. Moser, P. Melliar-Smith, and Y. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994. 29

[37] J. Dugan, S. Bavuso, and M. Boyd. Dynamic Fault Tree Models for Fault Tolerant Computer Systems. *IEEE Transactions on Reliability*, 41(3):363–377, 1992. 10

[38] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 411–420. IEEE, 1999. 46

[39] T. Eiter and T. Lukasiewicz. Complexity Results for Structure-Based Causality. *Artificial Intelligence*, 142(1):53–89, 2002. 91

[40] T. Eiter and T. Lukasiewicz. Causes and Explanations in the Structural-Model Approach: Tractable Cases. *Artificial Intelligence*, 170(6-7):542–580, 2006. 91

[41] Esterel Technologies. SCADE `http://www.esterel-technologies.com/products/scade-suite/`. 9

[42] L. Filippidis and H. Schlingloff. Structural Equation Modelling for Causal Analysis Applied to Transport Systems. In *Proceedings of the Ninth Symposium on Formal Methods (FORMS / FORMAT 2012)*, 2012. 9

[43] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, 1960. 70

[44] G. Gössler, D. L. Métayer, and J.-B. Raclet. Causality Analysis in Contract Violation. In *Proceedings of the First International Conference on Runtime Verification (RV 2010)*, volume 6418 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2010. 8

[45] 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):229–247, 2006. 8, 9

[46] J. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 54(6):843–897, 2005. 2, 8, 57, 58, 59

[47] T. Han, J.-P. Katoen, and B. Damman. Counterexample Generation in Probabilistic Model Checking. *IEEE Transactions on Software Engineering*, 35(2):241–257, 2009. 3, 10, 19, 107

[48] H. Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer, 2002. 10

[49] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. The PRISM Language - Semantics. Available from URL `http://www.prismmodelchecker.org/doc/semantics.pdf`. 21, 127

[50] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addision–Wesley, 2003. 14, 17, 92

[51] IEEE. IEEE STD 1028-2008 Standard for Software Reviews, 2008. 1

[52] International Council on Systems Engineering (INCOSE). Systems Modelling Language, Specification 1.2, Jun. 2010. 3, 27, 167

[53] International Electrotechnical Commission. Analysis Techniques for System Reliability - Procedure for Failure Mode and Effects analysis (FMEA), IEC 60812, 1991. 1, 170

[54] International Electrotechnical Commission. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, IEC 61508, 1998. 1, 7, 169

[55] International Organization for Standardization. Road Vehicles – Functional Safety, ISO 26262, 2011. 1, 146, 167, 169

[56] N. Jansen, E. Ábrahám, M. Volk, R. Wimmer, J.-P. Katoen, and B. Becker. The COMICS tool–Computing Minimal Counterexamples for DTMCs. In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA 2012)*, volume 7561 of *Lecture Notes in Computer Science*, pages 349–353. Springer, 2012. 108

[57] S. Kleinberg. *Causality, Probability, and Time*. Cambridge University Press, 2012. 55

[58] D. Kroening and O. Strichman. Efficient Computation of Recurrence Diameters. In *Proceedings of the Fourth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2003. 145

[59] V. G. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, Ltd., London, UK, 1995. 18

[60] T. Kumazawa and T. Tamai. Counterexample-Based Error Localization of Behavior Models. In *Proceedings of the Third International Symposium NASA Formal Methods (NFM 2011)*, volume 6617, pages 222–236. Springer, 2011. 8, 9

[61] M. Kuntz, F. Leitner-Fischer, and S. Leue. From Probabilistic Counterexamples via Causality to Fault Trees. In *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2011)*, volume 6894 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2011. 5, 29, 55, 107, 151

[62] M. Kuntz, F. Leitner-Fischer, and S. Leue. From Probabilistic Counterexamples via Causality to Fault Trees. Technical Report soft-11-02, Chair for Software Engineering, University of Konstanz, Germany, 2011. Available from `http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-11-02.pdf`. 5, 29, 55, 107, 151

[63] M. Kwiatkowska, G. Norman, and D. Parker. Controller Dependability Analysis By Probabilistic Model Checking. *Control Engineering Practice*, 15(11):1427–1434, 2007. 95

[64] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic Model Checking. In *Proceedings of the Seventh International School on Formal Methods for the Design of Computer, Communications and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *Lecture Notes in Computer Science (Tutorial Volume)*, pages 220–270. Springer, 2007. 17

[65] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. 18, 21

[66] M. Kwiatkowska, G. Norman, and D. Parker. The PRISM Benchmark Suite. In *Proceedings of the Ninth International Conference on Quantitative Evaluation of SysTems (QEST 2012)*, pages 203–204. IEEE CS Press, 2012. 95

[67] A. Laarman, J. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *Proceedings of the 10th Conference on Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 247–256. FMCAD Inc, IEEE, 2010. 88

[68] P. B. Ladkin. Root Cause Analysis: Terms and Definitions, Accimaps, MES, SOL and WBA. Available from URL `http://www.rvs.uni-bielefeld.de/publications/Papers/LadkinRCAoverview20130120.pdf`, February 2013. 165

[69] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. 16

[70] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21:558–565, July 1978. 8

[71] F. Leitner-Fischer and S. Leue. Quantitative Analysis of UML Models. In *Proceedings of Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2011). Dagstuhl, Germany.*, 2011. 27

[72] F. Leitner-Fischer and S. Leue. QuantUM: Quantitative Safety Analysis of UML Models. In *Proceedings of the Ninth Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, volume 57 of *EPTCS*, pages 16–30, 2011. 3, 27, 92, 118, 135, 167

[73] F. Leitner-Fischer and S. Leue. The QuantUM Approach in the Context of the ISO Standard 26262 for Automotive Systems (Extended Abstract). Technical Report soft-11-01, Chair for Software Engineering, University of Konstanz, Germany, 2011. 5, 6, 167

[74] F. Leitner-Fischer and S. Leue. Causality Checking for Complex System Models. Technical Report soft-12-02, Chair for Software Engineering, University of Konstanz, Germany, 2012. Available from `http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-12-02.pdf`. 5, 67

[75] F. Leitner-Fischer and S. Leue. Towards Causality Checking for Complex System Models. In *Proceedings of Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2012). Dagstuhl, Germany.*, 2012. 5, 67

[76] F. Leitner-Fischer and S. Leue. Causality Checking for Complex System Models. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, volume 7737 of *Lecture Notes in Computer Science*, pages 248–267. Springer, 2013. 5, 55, 67

[77] F. Leitner-Fischer and S. Leue. On the Synergy of Probabilistic Causality Computation and Causality Checking. In *Proceedings of the International SPIN Symposium on Model Checking of Software (SPIN 2013)*, volume 7976

of *Lecture Notes in Computer Science*, pages 246–263. Springer Verlag, 2013. 5, 55, 125

[78] F. Leitner-Fischer and S. Leue. Probabilistic Fault Tree Synthesis using Causality Computation. *International Journal of Critical Computer-Based Systems (IJCCBS)*, 4(2):119–143, 2013. 5, 29, 55, 107, 151

[79] F. Leitner-Fischer and S. Leue. Synergy of Probabilistic Causality Computation and Causality Checking. Technical Report soft-13-01, Chair for Software Engineering, University of Konstanz, Germany, 2013. Available from `http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-13-01.pdf`. 5, 125

[80] F. Leitner-Fischer and S. Leue. SpinCause: A Tool for Causality Checking. In *Proceedings of the International SPIN Symposium on Model Checking of Software (SPIN 2014)*, pages 117–120. ACM, 2014. 5, 67

[81] D. Lewis. *Counterfactuals*. Wiley-Blackwell, 2001. 2, 8, 55

[82] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., 1992. 15

[83] Z. Manna and H. Sipma. Alternating the Temporal Picture for Safety. In *Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 429–450. Springer Berlin Heidelberg, 2000. 24

[84] D. Muller, A. Saoudi, and P. Schupp. Weak Alternating Automata Give a Simple Explanation of Why Most Temporal and Dynamic Logics are Decidable in Exponential Time. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS 1988)*, pages 422–427, 1988. 25, 52

[85] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part I. *Theoretical Computer Science*, 13(1):85–108, 1981. 8

[86] Object Management Group. Unified Modelling Language, Specification 2.4.1, 2010. 3, 27, 167

[87] Object Management Group. XML Metadata Interchange, Specification 2.4.1, 2011. 27

[88] F. Ortmeier, W. Reif, and G. Schellhorn. Formal Safety Analysis of a Radio-Based Railroad Crossing Using Deductive Cause-Consequence Analysis (DCCA). In *Proceedings of the Fifth European Dependable Computing Conference (EDCC 2005)*, volume 3463 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 2005. 9

[89] C. A. Petri. Introduction to General Net Theory. In *Net Theory and Applications*, pages 1–19. Springer, 1980. 8, 160

[90] C. Power and A. Miller. Prism2Promela. In *Proceedings of the Fifth International Conference on Quantitative Evaluation of Systems (QEST 2008)*, pages 79–80. IEEE, 2008. 127

[91] Radio Technical Commission for Aeronautics (RTCA). DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification, 2011. 1

[92] G. Schellhorn, A. Thums, and W. Reif. Formal Fault Tree Semantics. In *Proceedings of the Sixth World Conference on Integrated Design & Process Technology (IDPT 2002)*. Society for Design and Process Science, 2002. 55

[93] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An Interval Logic for Higher-Level Temporal Reasoning. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC 1983)*, pages 173–186. ACM, 1983. 29

[94] G. S. Shedler and G. Shedler. *Regenerative stochastic simulation*. Academic Press New York, 1993. 18

[95] Siemens AG. Siemens Norm SN29500, Ausfallraten Bauelemente. 3, 107, 118

[96] Telcordia Technologies. Sr-332 Reliability Prediction Procedure for Electronic Equipment, 2001. 3, 107, 118

[97] U.S. Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492. 1, 3, 22, 23, 151, 164

[98] M. Vardi. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996. 23, 24

[99] M. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1):1–37, 1994. 25, 52

[100] S. Wang, A. Ayoub, B. Kim, G. Gössler, O. Sokolsky, and I. Lee. A Causality Analysis Framework for Component-Based Real-Time Systems. In *Proceedings of the Fourth International Conference on Runtime Verification (RV 2013)*, volume 8174 of *Lecture Notes in Computer Science*, pages 285–303. Springer, 2013. 8

[101] R. Wimmer, N. Jansen, E. Ábrahám, B. Becker, and J.-P. Katoen. Minimal Critical Subsystems for Discrete-Time Markov Models. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, volume 7214 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2012. 10

[102] G. Winskel. *Event Structures*. Springer, 1987. 8, 160

[103] H. Younes. Ymer: A Statistical Model Checker. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 429–433. Springer, 2005. 18

[104] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Elsevier, 2009. 8, 55