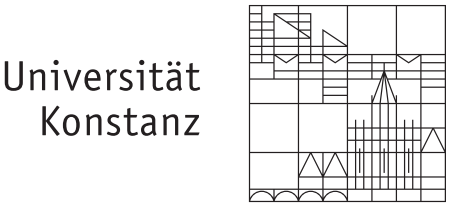
Causal Analysis and Repair of Systems

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

Presented by

Martin Kölbl

at the



Faculty of Sciences
Department of Computer and Information Science

Konstanz, 2022

Dissertation of the University of Konstanz

Date of the oral examination: January 11th 2023

1. Referee: Prof. Dr. Stefan Leue, University of Konstanz, Germany
2. Referee: Prof. Dr. Thomas Wies, New York University, USA
3. Referee: Prof. Dr. Falk Schreiber, University of Konstanz, Germany

Acknowledgments

Writing a dissertation is a long and rocky journey and I want to thank several people for the support on this journey.

Most of all, I want to thank Stefan Leue for his outstanding guidance in research and patience with writing papers. As a result, we traveled around the world to spread our ideas. We had an international collaboration with Thomas Wies. He gave me valuable advice, and I also want to thank him for being the second referee.

I also thank Georgiana Caltais who was a cooperative colleague. In addition, I want to thank Hargurbir Singh for being a good fellow, for the joy he spread, and the good stories about India. Some pleasant work days were spent with him, Stefan Brütsch and Ahmed Eroglu.

My special thanks also goes to my friends Eva Schiebel and Christian Fischer for the many dinners with good discussions.

And finally, my gratitude goes to my beautiful wife Sarah Stoll for listening to my little problems for hours. You showed me the best roller coasters of life, your presence brings me back to balance and in the remaining time you put a smile on my face.

Abstract

A system consists of hardware and software components and makes life-critical decisions, for example, in a pacemaker. The interactions between the components of a system lead to a high degree of concurrency, which makes it a complex task to verify whether a system model violates a property of its specification. Different approaches exist to verify whether a system satisfies its specification. *Test procedures* are often used but are labor intensive and time-consuming. In contrast, *Model checking* tools systematically search through the state-space of a system model to verify whether a property in the specification is violated. When a model checker finds a property violation, it usually returns a counterexample in the form of an execution that leads to this property violation. Model Checkers are even more efficient in finding property violation than testing procedures [28].

Realistic systems also consist of hardware components that can fail, for instance, due to fatigue of material. In these systems, a property violation cannot always be prevented. Safety standards such as ISO26262 [76] recommend to justify by *safety cases* that a property violation is sufficiently improbable, and therefore the system is acceptably dependable. For a safety analysis, it does not suffice to find a single counterexample, but an enumeration of the possible property violations is needed.

Causality Checking [104] is an algorithmic method that utilizes a model checker to enumerate causes that explain every property violation in a model. In Causality Checking, a cause for a property violation is based on the *counterfactual conditional* [108] that if the cause occurs, the property violation occurs, and if the cause would not have occurred, the property violation would not have occurred. In the analysis, Causality Checking compares the action sequences in the executions of a system model and computes causes in the form of ordered actions. Causality checking is implemented in the tool *QuantUM*, which analyzes the *SysML* model [115] of a system, computes causes for the property violations of the system, and represents these causes in a *fault tree* [134]. Causality checking is a promising approach to support the safety analysis of systems.

The first significant contribution of this work is to extend and improve Causality Checking. For a safety analysis, it is essential to compute every possibility of the system to result in a property violation. Therefore, we propose revised cause definitions and analyses compared to [102] that are complete and compute each cause for a property violation in a system. Furthermore, we explicitly compute the action order in a cause, which has not been done before. The results show that these order relations help to understand the occurrence of a property violation and the difference between causes where the action sets are subsets of one another. We demonstrate that Causality Checking can support the safety analysis by a case study considering an autonomous driving assistant. Contrary to manual driving, an autonomous driving assistant cannot rely on a driver and needs to be fail-operational, which means that the function to drive a car is provided, even in the event of a system failure. Therefore, we propose also a method to analyze whether the system is fail-operational. The results show that Causality Checking

supports the safety analysis of a system. Furthermore, the automated nature of the presented analyses allows a developer to compare and evaluate design decisions early in the development of a system.

The second significant contribution of this work is to adapt the idea of causes to real-time models. In an embedded system, such as a pacemaker, time is an essential aspect for the correctness of the system. We can describe such a timed system by a real-time model [7] where time passes continuously within a location of the model and not explicitly by transit to another location. The *timed counterexample* of a real-time model contains a sequence of actions and time constraints that limit the possible delay values within a location. An assignment of every delay value in the timed counterexample is an *execution*. Considering that for the same timed counterexample, one execution can lead to a property violation and another execution can not. Both executions contain the same sequence of actions and differ only in the chosen delay values. Hence, Causality Checking cannot compute a cause for this violation, since it analyzes only the action sequences in an untimed system.

In this thesis, we propose a dynamic and a static analysis that analyze a timed counterexample and compute causes for a property violation. The dynamic analysis compares the possible delay values in a timed counterexample. It computes causal ranges in which every delay assignment leads to a property violation and an alternative value assignment exists that does not lead to a property violation. Accordingly, the counterfactual conditional is satisfied and the causal range is causal.

The static analysis is a repair method that computes alternative time constraints for a timed counterexample such that the property violation is no longer reachable. Since alternate time constraints prevent the property violation that occurs with the original time constraints, the counterfactual conditional is fulfilled and the original time constraints can be assumed as causal for the property violation. These alternative time constraints are potential repairs for the real-time model underlying the timed counterexample. The alternative time constraints affect the possible executions of the real-time model, and therefore may change the functional behavior of the real-time model. We propose an admissibility test to ensure that the altered time constraint does not change the functional behavior of the real-time model and is, therefore, a repair for the real-time model. We apply both the dynamic and static analysis to several real-time models from literature and show that the computed causes help to understand the occurrence of property violations and to prevent violations in real-time models.

The presented approaches automatically compute causes for the occurrence of system failures in systems with and without real-time aspects. Integrating these approaches into an Model-driven engineering (MDE) development process may save manpower and may make systems more dependable.

Zusammenfassung

Ein System besteht aus Hardware- und Softwarekomponenten und trifft beispielsweise in einem Herzschrittmacher lebenswichtige Entscheidungen. Um die Verlässlichkeit eines Systems zu gewährleisten, wird in einer Spezifikation beschrieben, welche Eigenschaften das System einhalten muss. Die Komponenten in einem System agieren häufig in einem hohen Maß gleichzeitig, weshalb es eine komplexe Aufgabe darstellt zu überprüfen, ob das System seine Eigenschaften einhält. Es gibt verschiedene Ansätze, um festzustellen, ob ein System seiner Spezifikation entspricht. Die zumeist eingesetzten Testverfahren sind arbeitsintensiv und zeitaufwendig. Im Unterschied dazu durchsucht ein *Model Checker* systematisch den Zustandsraum eines Systemmodells, um zu überprüfen, ob eine Eigenschaft verletzt wird. Falls ein Model Checker eine Eigenschaftsverletzung findet, gibt er in der Regel ein Gegenbeispiel in Form einer Ausführung zurück, die zu dieser Eigenschaftsverletzung führt. Beyer und Lemberger konnten zeigen, dass Model Checker Eigenschaftsverletzungen effizienter finden als Testverfahren [28].

Reale Systeme enthalten jedoch auch Hardwarekomponenten, die zum Beispiel verschleißbedingt ausfallen können, weshalb in diesen Systemen das Auftreten einer Eigenschaftsverletzung nicht verhindert werden kann. Sicherheitsstandards wie ISO26262 [76], empfehlen durch *Safety Cases* zu begründen, dass Eigenschaftsverletzungen ausreichend unwahrscheinlich sind und daher das System hinreichend verlässlich ist. Um die Verlässlichkeit eines Systems nachzuweisen, ist es notwendig, nicht nur ein Gegenbeispiel zu finden, sondern sämtliche Möglichkeiten einer Eigenschaftsverletzung aufzuzeigen.

Causality Checking [104] ist eine algorithmische Methode, die einen Model Checker nutzt, um die Gründe für Eigenschaftsverletzungen in einem Modell ohne zeitlichen Aspekt aufzuzählen. In Causality Checking basiert ein Grund für eine Eigenschaftsverletzung auf dem *kontrafaktischen Konditional* [108], das aussagt, dass beim Auftreten des Grundes die Eigenschaftsverletzung eintritt, und andererseits bei Ausbleiben des Grundes auch die Eigenschaftsverletzung nicht eingetreten wäre. In der Analyse vergleicht Causality Checking die Aktionsfolgen in den Ausführungen eines Systemmodells und berechnet die Gründe in Form von teilweise geordneten Aktionen. Causality Checking ist in dem Tool *QuantUM* implementiert. Um ein System zu analysieren, wird QuantUM ein SysML-Modell [115] des Systems und eine Spezifikation übergeben. Für dieses Modell berechnet QuantUM dann Gründe für die Erreichbarkeit von Eigenschaftsverletzungen im System und stellt diese in der Form eines *Fehlerbaums* [134] dar. Causality Checking ist ein Ansatz, um die Sicherheitsanalyse von Systemen zu unterstützen.

In dieser Arbeit wird Causality Checking erweitert und verbessert. Die zugrundeliegenden Ideen werden auch auf Echtzeitmodelle angewendet.

Für eine Sicherheitsanalyse ist es wesentlich, alle Gründe für eine Eigenschaftsverletzung zu berechnen. Deshalb schlagen wir Kausalitätsdefinitionen und Analysemethoden vor, die vollständig sind und jeden Grund für eine Eigenschaftsverletzung berechnen. Außerdem berechnen wir für einen Grund explizit

die kausale Reihenfolge zwischen Aktionen. Die Evaluationsergebnisse zeigen, dass diese kausalen Reihenfolgen helfen, das Auftreten einer Eigenschaftsverletzung und den Unterschied zwischen verschiedenen Gründen zu verstehen. Wir zeigen durch eine Fallstudie über einen autonomen Fahrassistenten, wie Causality Checking eine Sicherheitsanalyse unterstützen kann [86]. Im Gegensatz zu pilotiertem Fahren, gibt es bei einem autonomen Fahrassistenten keinen Fahrer, der in einem Fehlerfall die Situation bewältigt. Daher muss ein autonomer Fahrassistent ausfallsicher sein. Dies bedeutet, dass die Fahrfunktion eines Autos auch im Falle einer Eigenschaftsverletzung noch ausreichend lange gewährleistet ist, sodass der Fahrassistent das Auto zum Beispiel auf den Seitenstreifen fahren kann. Für die Sicherheitsanalyse des autonomen Fahrassistenten nutzten wir Causality Checking und schlagen eine Erweiterung vor, um die Ausfallsicherheit des Systems zu analysieren. In der Fallstudie analysieren wir verschiedene Varianten eines Fahrassistenten. Da die Analysen keiner manuellen Interaktion bedürfen, erlauben sie einem Entwickler, die Auswirkung von Designentscheidungen mit einem geringen Arbeitsaufwand frühzeitig in der Systementwicklung zu vergleichen.

In einem eingebetteten System, wie einem Herzschrittmacher, ist Zeit ein wesentlicher Aspekt für die Korrektheit des Systems. Daher wenden wir die Ideen einer kausalen Analyse auf Echtzeitmodelle an. Zeit vergeht in einem Echtzeitmodell [7] kontinuierlich innerhalb eines Zustandes und stellt keinen expliziten Übergang zu einem anderen Zustand dar. Das *zeitliche Gegenbeispiel* eines Echtzeitmodells enthält eine Sequenz an Aktionen und zeitlichen Beschränkungen, die die möglichen Verzögerungen innerhalb von Zuständen einschränken. Wird allen Verzögerungen in einem zeitlichen Gegenbeispiel ein Wert zugewiesen, stellt dies eine Ausführung des Echtzeitmodells dar. Für dasselbe zeitliche Gegenbeispiel kann eine Ausführung zu einer Eigenschaftsverletzung führen und eine andere Ausführung nicht. Causality Checking vergleicht Ausführungen durch die Reihenfolge von Aktionen, die aber in den beiden Ausführungen des Echtzeitmodells gleich sind. Daher kann Causality Checking keinen Grund finden, warum eine der Ausführungen zu einer Eigenschaftsverletzung führt und die andere nicht.

In dieser Arbeit diskutieren wir eine dynamische und eine statische Analyse, die jeweils ein zeitliches Gegenbeispiel analysieren und Gründe für das Auftreten einer Eigenschaftsverletzung berechnen. Die dynamische Analyse vergleicht die mögliche Wertzuweisung der Verzögerungen. Sie berechnet kausale Wertebereiche für diejenigen Wertzuweisungen, die für Werte in diesem Bereich immer zu einer Eigenschaftsverletzung führen und für die eine alternative Wertzuweisung außerhalb des Wertebereiches nicht zu einer Eigenschaftsverletzung führt. Entsprechend ist das kontrafaktische Konditional erfüllt und der kausale Wertebereich stellt einen Grund für die Eigenschaftsverletzung dar.

Die statische Analyse ist eine Reparaturmethode, die für ein zeitliches Gegenbeispiel alternative zeitliche Beschränkungen berechnet, welche die Eigenschaftsverletzung verhindern. Da das zugrundeliegende Modell mit den alternativen zeitlichen Beschränkungen die Eigenschaftsverletzung nicht erreicht, ist das kontrafaktische Konditional erfüllt und die jeweils ursprüngliche zeitliche Beschränkung ist daher auch ein Grund für die Eigenschaftsverletzung. Eine alternative zeitliche Beschränkung stellt gleichzeitig eine Reparatur für das zugrundeliegende Echtzeit-

modell dar. Eine Reparatur beeinflusst die möglichen Ausführungen des Modells und kann infolgedessen das funktionale Verhalten des Echtzeitmodells verändern. Wir schlagen daher eine Zulässigkeitsprüfung vor, die für eine Reparatur überprüft, ob das funktionale Verhalten des Echtzeitmodells nicht verändert wird und eine zulässige Reparatur ist. Wir wenden sowohl die dynamische als auch die statische Analyse auf mehrere Echtzeitmodelle aus der Literatur an und zeigen, dass die berechneten Gründe helfen das Auftreten einer Eigenschaftsverletzung zu verstehen und das Echtzeitmodell zu reparieren.

Die vorgestellten Methoden nutzen Kausalität, um das Auftreten von Fehlern in einem System mit oder ohne zeitlichem Aspekt zu erklären. Mögliche Fehler in einem System zu verstehen, ist eine Vorraussetzung dafür die Verlässlichkeit des Systems zu erhöhen. Die Methoden unterstützen eine automatisierte Fehleranalyse eines Systems und können in einen modellgetriebenen Entwicklungsprozess integriert werden.

Contents

Part I

Introduction

Chapter 1

General Introduction

1.1 Motivation

Software is used in messengers to communicate online, is hidden in the electronic hardware of systems, and even makes life-critical decisions, for instance, in a pacemaker. We continuously integrate more complex systems in our daily life, for example, by improving the driving assistants in our cars. Our daily routines rely on the dependability of such systems. In order to ensure dependability, a system has to behave as expected. The expected behavior is commonly defined by requirements. When a system deviates from its requirements, it fails [10]. Such a failure can lead to substantial financial losses or, in the case of a *safety-critical system*, harm persons. A system is dependable, if we can ensure that either a failure cannot happen or the probability of every potential failure is below a certain threshold [10]. Since systems continuously integrate more functionality and cooperate with other systems, the task to ensure dependability becomes more complex. In the following, we present promising methods to deal with this complexity.

Model-driven engineering (MDE) [38, 130] uses a system model to reduce the complexity of a system and to automate development tasks whenever possible. A system consists of hardware and software components. A pure software model can be described in the *Unified Modeling Languages (UML)* [116]. With the derivate *Systems Modeling Language (SysML)* [115], system models including hardware can be described. The languages include diagrams that semi-formally describe the static structure and the dynamic behavior of a system. A system usually consists of several components, which can be depicted in SysML by a block definition diagram. The behavior of each system component can be described by a state-machine diagram (STM). An STM depicts possible states, transitions between these states, and actions that trigger the transitions to change the state in a component. Several transitions in different STMs can be enabled simultaneously. This makes it possible to take the transitions in different orders, which leads to different system interleavings. Since every transition is labeled with an action, we can also describe an interleaving of transitions by its sequence of actions and call it an *action trace*. Every interleaving is a possible execution of the system, or in other words, a possible behavior. We refer to these behaviors as the functional behavior of the system. Real-time can be also important in order to ensure the correctness of a system. In the Pacemaker model [80], for instance, it is life-critical that a pace happens periodically within a specific time. An extension of SysML to describe real-time constraints is MARTE [117].

A model checking algorithm [14] systematically analyzes the behavior of a system to ensure its correctness with respect to a given specification. Otherwise, it will

detect a state that violates a desired property of the system. In case a failure is found, the model checking algorithm usually returns a counterexample in the form of an execution that leads to the failure. Although a model checker finds failures more efficiently than testing [28], model checkers are typically not used in system development. One reason is that model checkers use their own input languages and, hence, can not be easily integrated into MDE [109]. This problem can be tackled by automated transformations that convert SysML to the input languages of model checkers. Automated transformations are described, for instance, for SysML models in [30,90] and for SysML models with real-time aspects in [18,71]. The main challenges to use a model checker to analyze a system are:

- If a model checker finds a counterexample, the counterexample consists of many transitions that describe a complex interaction between several system components. On the one hand, the counterexample contains a specific cause, since otherwise, the failure would not be reached. On the other hand, not every transition of a counterexample may be required to reach the failure state. An additional analysis is necessary that computes why certain behaviors in the system can reach a property violating system state.

- A realistic system can contain unpreventable failures like hardware failures. For a system with unpreventable failures, a dependability analysis needs to be based on a technique that does not only witness a single failure, but that enumerates the possibilities of a system to fail. Standards such as ISO26262 [76] then cover every failure possibility by a safety case to ensure that the system is acceptably safe.

Causality Checking [102,104] is an approach that overcomes these challenges. It uses a model checker to find counterexamples in a system model and analyzes the contained action traces to enumerate the causes for reaching property violations. The underlying cause definition is an adaption of actual causality [59,60] to system models. A cause describes a minimal set of actions and their order such that the failure occurs (regularity) whereas with fewer actions or in a different order the failure would not occur (counterfactual). Causality Checking is implemented in the QuantUM tool [103] and depicts the causes in a fault tree [134]. Fault trees are well known by engineers who are not experts in model checking. In dependability analysis, safety cases are created that consider every cause in the fault tree.

Causality Checking as proposed in [102, 103] uses a cause definition that is incomplete, as we will see in Section 4.2. Additionally, this cause definition cannot compute a cause when the same action trace in a system model non-deterministically violates and satisfies a property. Such an action trace violates the cause condition that every action trace that is according to a cause has to violate the property. For a dependability analysis, we need a complete version of Causality Checking. Otherwise, a possibility of the system to fail may not be covered by a safety case. Therefore, we propound complete cause definitions and algorithms to compute every cause in a system model. In line with the cause definition originally proposed, we consider the actions and their ordering in a counterexample to be causal for a failure to occur. We also propose algorithms that explicitly compute the action orders in a cause and depict these orders in a fault tree.

We show by a case study on autonomous driving that this enhanced version of Causality Checking can support the development of a safety critical system. Driver assistance systems in vehicles are safety critical systems of high complexity, which motivates the usage of formal analysis techniques [132], such as Causality Checking. As a case study, we analyze with Causality Checking the safety of an autonomous driving vehicle architecture to support the recommendations of the the ISO 26262 standard [76]. The standard describes recommendations for the development of safe road vehicles. However, these recommendations rely on the presence of a driver who is not present in an autonomous driving system. Accordingly, we present a safety analysis in the sense of the ISO 26262 standard that follows the recommendations whenever possible. In the design of a system architecture, it is important to evaluate the possible design decisions. Making design decisions differently results in a different version of an architecture and the implications of the chosen design decisions are not obvious from the outset. We demonstrate how Causality Checking can automate the analysis of several architecture versions. Differences in the computed causes make the implications of the design decisions explicit. In this way, the impact of design decisions is understood and can be carefully considered at an early stage of system development.

Causality Checking can only analyze untimed systems and does not compute timed causes. In order to overcome this limitation, we describe a timed system by a real-time model where time progresses while the model is in a concrete location. Real-time model checkers, such as UPPAAL [26] and oppaal [39], find a failure in a real-time model, if any, and return a timed counterexample in this case. We refer to a timed counterexample also as timed diagnostic trace (TDT). A TDT is a sequence of actions that leads to a failure and is annotated with time constraints that bound the possible values of time delays in a state. Since a TDT can describe multiple possible delays for a state, a TDT is symbolic and describes a variety of executions. For the same TDT, one choice of the delay values can lead to a failure while another choice of the delay values may not lead to a failure. Causality Checking could be applied straight forward by removing time constraints in the TDTs and causally analyzing only the action traces in TDTs. Obviously, this approach cannot explain in a TDT, why for some delay values, the TDT reaches a failure and for other delay values it does not. The cause for the violation is not within the action sequence but the assigned delay values.

The assignment of a delay value in a TDT is influenced dynamically by the choice for a specific value and statically by the time constraints that restrict the possible assignments to a delay. Since both influences can be considered causal, we present a dynamic and a static timed cause analysis. The dynamic analysis considers the assignment of delay values as causal for reaching a failure, since choosing a different assignment prevents the violation, which satisfies the counterfactual argument. As such, the dynamic analysis computes a cause by comparing the possible behaviors allowed in the system model similar to Causality Checking. The static analysis searches for alternative time constraints in the TDT that prevent the failure. As an alternative constraint prevents the violation, the static analysis also satisfies the counterfactual argument, and we determine the original constraint as causal for the violation. The alternative constraint represents also a potential repair that

corrects the underlying real-time model. However, it may also be too strict and remove functional behavior from the real-time model. Therefore, we provide an admissibility test to ensure that the repair does not change the functional behavior of the real-time model. In order to evaluate the analyses, we applied them to a set of TDTs that we created for various real-time models from the literature. The dynamic analysis computes causal delay ranges that explain when a real-time model fails. The static analysis computes at least one admissible repair for 69% of the TDTs. These repairs can be directly applied to correct a real-time model.

In summary, we propose approaches to causally analyze the functional behavior and the real-time aspect of a system model. The computed causes explain why a system fails. This is a necessary insight, for instance, to repair a faulty system and to ensure system dependability.

1.2 Contributions

We modify and extend Causality Checking for untimed systems and propose causal analyses that consider the real-time aspects in timed systems.

In more detail, we propose the following modifications on Causality Checking for untimed systems in Part II:

1. complete cause definitions for Causality Checking,
2. an algorithm that completely enumerates the action traces required to compute the causes in a system model, and
3. an analysis to compute and depict the partial event orders in a cause.

We evaluate this advanced version of Causality Checking by a case study on the safety analysis of an automated driving architecture.

In Part III, we present approaches to causally analyze the real-time aspect in TDTs:

1. Approach analyzes in a TDT which delay assignments are causal for reaching a property violation.
2. Approach analyzes the time constraints in a TDT to compute repairs that prevent the failure. We also propose an admissibility test to check whether a repair is also admissible in the overall real-time system.

1.3 Outline

Chapter 2 contains an overview of the current state of the art and related work.

Chapter 3 presents necessary foundations and provides basic definitions.

Chapter 4 proposes revised cause definitions for untimed systems.

Chapter 5 presents an algorithm to compute every action trace that leads to a failure in an untimed system.

Chapter 6 shows algorithms to analyze the strict partial order in causes and depict these orders by a fault tree.

Chapter 7 exemplifies how Causality Checking can support the safety analysis of an automated driving architecture according to the ISO26262.

Chapter 8 gives a formalization of a TDT that we use for the analyses in the following chapters.

Chapter 9 sets up a dynamic analysis to compute delay values that are causal for the property violation in a TDT.

Chapter 10 presents static repair analyses for the time constraints in a TDT that compute admissible repairs for a timed model.

1.4 Attribution of Contributions

Parts of this thesis have already been published in [86–89, 91, 92]. Stefan Leue coauthored every of the beforehand mentioned papers. Florian Leitner-Fischer first presented research about Causality Checking in [102–105] and its implementation in the tool QuantUM. At the beginning of this thesis, we applied Causality Checking to analyze the architecture of an automated driving vehicle in [86]. We detected an under-approximation in the cause definition and the cause computation of Causality Checking, and presented an efficient algorithm that overcomes the algorithmic approximation in [87]. In the original version, QuantUM already depicted the causes of a model by a fault tree but did not depict the possible action orders within the execution of a cause. We extended QuantUM by the possibility to depict the action orders in [88]. The work published in [87] is presented in Chapter 5 and the work published in [88] is presented in Chapter 6. The results of [88] motivated the revision of the fault trees published in [86] by fault trees that also depict the action order. The preliminaries of these papers are also part of Chapter 3.

In parallel to the causal analysis of untimed systems we started to analyze the time aspect in the TDT of a real-time system. We present an algorithm in [89] that analyzes the time delays in a TDT and computes the time delay values that are causal for the property violation. The work published in [89] is presented in Chapter 9. In collaboration with Thomas Wies, we presented in [91–93] analyses to compute syntactic repairs that prevent the property violation in a TDT, and to check whether the computed repairs are admissible in the real-time system. Thomas Wies also coauthored these papers. The work published in [91–93] is presented in Chapter 10 and the formal trace encoding is part of Chapter 8. This work on computing repairs resulted in the tool TarTar. The tool is publicly available at `https://github.com/sen-uni-kn/tartar`.

Chapter 2

State of the Art and Related Works

We focus on the computation of causes to explain the occurrence of failures in a system. In order to compute a cause for a system failure, we need a technique to detect a failure in a system. Techniques to find failures in a system are either categorized as testing or formal verification techniques. Testing relies on a fault hypothesis and can only witness the presence of a fault but not its absence. Model checking is a formal verification technique that searches the state space of a given model to verify whether the model is correct according to a given specification or not. Using a model checker to verify UML/SysML models is well researched [8, 32, 85, 99, 124, 135, 144] and is also applied to verifying timed systems [129]. The focus of those papers lies on the correct transformation from a UML/SysML model to a model checking language to verify, with the help of a model checker, whether the system model is correct. The result is a counterexample that witnesses the failure and facilitates debugging the underlying fault, but it does not give the reason for the failure. In this thesis, we propose several formal verification techniques that causally analyze counterexamples of untimed and timed systems to explain why a property violation is reachable.

2.1 Causal Analyses of Untimed Systems

Various approaches towards analyzing counterexamples in order to explain system failures exist. [15, 56, 57] use a model checker to find counterexamples in a program and compare several counterexamples with each other in order to find a cause. Similarly, the approaches in [22,107] statistically analyze the occurrence of actions to mine action patterns that probably cause system failures. The authors of [114, 143] use automated testing and minimize counterexample by comparing a non-failure and a failure execution of a program. None of these works enumerates the causes of a system model. These works do not explicitly rely on a causal theory, the authors ensure minimality and in some works also compare failure and non-failure executions with another just as in the works below that rely on a causal theory.

A typical distinction of causal theories is to differentiate between type causality and actual causality. In both theories, the cause must happen temporally before the effect. While type causality considers predictions such as “smoking causes lung cancer”, actual causality considers why a certain effect occurred in a given situation [61]. For example, assume a person called Susi threw a rock at a bottle, the rock hit the bottle and the bottle shattered. We now want to analyze whether Susi throwing a rock was the actual cause for the bottle to shatter. An actual cause

by [108] satisfies the counterfactual argument. This argument holds when in case Susi would not have thrown the rock, then the bottle would not have been shattered. As we cannot change reality after the fact, the counterfactual argument can only be checked for a given artificial model. A formal and rigorous definition of actual causality based on a suitable model called structural equation model (SEM) is presented in [59, 61, 62].

It is not trivial to check whether the counterfactual argument holds when multiple potential causes occur in a given model. Assume an extended version of the rock throwing example where Susi and another person called Bob throw rocks [59]. The throws are both perfect. So the first rock that hits the bottle also shatters the bottle. Susi's rock arrives first and shatters the bottle. Intuitively, we would assume the throw of Susi to be the cause for shattering the bottle. A straightforward test of the counterfactual argument fails: If Susi would not have thrown the rock, the bottle would still be destroyed because of Bob throwing a rock. Halpern and Pearl introduce contingencies in [59] to block Bob from throwing his rock and so Susi throwing a rock satisfies the counterfactual argument and is the actual cause in the extended rock example.

The relationship between an effect and its actual cause is refined in [19, 58] by dependency and production. The bottle shattering depends on Susi throwing a rock and Bob throwing a rock since without a throw the bottle would not shatter. Susi throwing the rock is also productive in the sense that her rock actually shatters the bottle. In an alternative scenario where Bob's rock reaches the bottle first, Susi's throw is not productive but Bob's throw is productive and, hence, the actual cause. An actual cause of a scenario relies on dependency and production. In this thesis, we propose cause analyses to enumerate the potential actual causes for a violation in a given system model. The violation depends on every computed cause. This means, even if one computed cause occurs in a given behavior it is still possible that another occurring cause can be productive and is therefore the actual cause. In this, the proposed cause analyses differ from the actual cause analysis in [59, 61, 62].

Early work on applying actual causality in the sense of [62] to the counterexample of a model checker is [21]. The approach analyzes a single counterexample and returns a set of causal variable values. [17] encodes actual causality in the sense of [62] in first order logic. Similarly, [73, 74] encode actual causality in first order logic to infer an actual cause for a property violation in acyclic models with binary variables. An overview of different approaches and techniques to explain model failures through causality is also given in [12]. We differ from those approaches in that we focus on approaches that enumerate each cause in a model that leads to a system failure.

The Computation of Every Cause in a System Model is addressed by several papers that search for all minimal counterexamples [20, 102, 104, 106]. The algorithm described in [102, 104, 106] computes an approximation as we will see in Chapter 5. The algorithm in [20] is also approximative since it is based on bounded model checking. None of the above mentioned approaches returns a complete set of counterexamples, which is necessary to completely enumerate causes. The approach in [31] uses process algebra to compute causal traces of a system model that vio-

lates a property in Hennessy Milner logic [67]. The cause definition differs from our work in the analyzed property language, and the explicit encoding of every trace of a model, which makes the approach rather inefficient.

Note that even though causes describe minimal action traces in this work, they differ from minimal length paths in graphs, which is why the vast literature on shortest path searches on graphs [47] and the computation of minimal length counterexamples [2, 64, 65, 128] are not directly applicable to the computation of actual causes.

This thesis is also related to work that enumerates a finite set of counterexamples without a reference to causality. Only a few works explicitly address the enumeration of counterexamples. A explicit state search based model checker does not have to terminate after finding the first counterexample and can continue to find further counterexamples. Such an approach does not need to terminate since a system model can contain infinite many counterexamples. An approach to enumerate all cycles that violate a liveness property is given in [140] but the approach does not return an explanation for the property violation. The approach in [43] computes single actions in programs that they determine as potentially responsible for a violation. An incomplete method to enumerate the k-shortest paths of a model and a survey on similar algorithms is given in [3].

Depicting and Computing the Action Order in a Set of Action Traces. The approach in [20, 102, 106] proposed Event Order Logic (EOL) [104] to express the action order in a cause, however, we are not aware of an automatized technique to compute an EOL formula from a set of action traces to express those action traces. A step in this direction is to analyze the partial order of actions in a set of action sequences. A set of arbitrary action sequences can be expressed by a set of partial orders. The computation of a minimal number of partial orders to express the set of sequences is known as the partial-order-set cover problem which is NP-complete [66]. In contrast to this approach, we compute a single partial order for each set of action traces with the same action set. A Mazurkiewicz trace [44] describes a set of action traces by a sequence t of events and an independence relation D. The D is symmetric, which means that when (a, b) is in D, then (b, a) is in D. In t, two neighboring actions a and b can be reordered when (a, b) is in D. In the context of causality, we are not interested in the independence relation between actions but in the order relation of a cause describing the necessary action orders to reach a property violation. Similarly, Lamport's happen-before relation describes strict partial orders, for instance, of messages in an asynchronous system [98]. A happen-before relation describes the possible ordering of the events that it contains, but does not describe the process that leads to a certain effect, as a cause does that we compute. A (strict) partial order is usually depicted by a Hasse diagram which is an undirected acyclic graph where the edges between lower vertices with vertices above describes a partial order relation [48]. In the context of Causality Checking, we prefer to use fault trees to depict a set of causes and the orders that the contained actions represent, since fault trees present a notation that is well known to engineers of safety-critical systems.

Safety Analysis of an Automated Driving System (ADS). The most closely related work on automated model-based safety analysis for autonomous vehicles is [54]. It uses a block definition diagram and a manually created fault tree to compute probabilities for the purpose of safety analysis. In contrast to our work, no causal explanations for failures are automatically derived from the model. An automotive driving system with a fail-operational mode is formally verified by a model checker in [125]. In a similar way, a case study in [52] verifies a vehicle control. Both approaches verify a system and do not compute causes or probabilities for a failure to occur. Model-based techniques are applied to evaluate an automotive architecture also in the following papers. The approaches of [36, 126] need a manual creation of a model in the model checker language and do not address the specifics of ADS. UML models, which are similar to SysML models, are also verified in [131] and [11], but both do not quantify the probability of system failures to occur. The analysis framework in [84] can analyze automotive systems with real-time aspects and probabilities. Also, it generates test cases to check that an implementation conforms with the model behavior. A cause computation or failure enumeration is not part of the framework. The work described in [1] addresses safety engineering for autonomous vehicles. It proposes no formal verification technique. Instead, it gives an overview of the current state of the art to develop and validate a safe automated driving system.

2.2 Analysis & Repair in Timed Systems

Causal reasoning is considered for real-time systems in [55, 136] and for reactive systems in [46]. In these three approaches, a system consists of several black-box components and the analysis blames failing components for a fail of the system. In contrast, we propose approaches to analyze the delay assignments and time constraints in a TDT.

In the following, we present the relatively few existing results available on a formal treatment of TDTs. The zone-based approach to real-time model checking, which relies on a constraint-based abstraction of the state space, is proposed in [68]. The use of constraint solving to perform reachability analysis for NTAs is described in [141]. This approach ultimately leads to the on-the-fly reachability analysis algorithm used in UPPAAL [27]. [45] defines the notion of a time-concrete UPPAAL counterexample. Work documented in [121] describes the computation of concrete delays for symbolic TDTs. The above cited approaches address neither fault analysis nor repair for TDTs.

The Analysis of Time Delays in a TDT is one approach to compute a cause for a property violation in a timed system. We are not aware of any work to compute causal time delays. The most appropriate example in actual causality for the causal analysis of time delays is an election [61]. Assume an election with 11 voters where a majority of yes votes results in an event to occur, otherwise the event does not occur. In case 7 votes are in favor of the event to occur, every single vote of these 7 votes is responsible for the event to happen. However, no voter is fully responsible

since the event would have also occurred when one voter had a different opinion. Similarly, failures exist in a timed system where several time delays contribute to miss a time bound and reaching a system failure but none of the single delays is fully responsible for the failure.

The Computation of a Timed Repair is another approach to analyze a TDT. Our repair analyses use MaxSMT solvers to compute minimal repairs. It is inspired by the use of MaxSAT solvers for fault localization in C programs which was first explored in the BugAssist tool [81]. Our approach also shares some similarities with syntax-guided synthesis [5, 122], which has also been deployed in the context of program repair [100]. One key difference is how we determine the admissibility of a repair in the overall system, which takes advantage of the semantics restrictions imposed by timed automata. In our approach, we repair timed systems that violate a property by reaching some error state, and we compute repairs that modify time constraints to prevent the TDT to reach a failure. While we modify constraints of a timed system, the repairs computed in [49] introduce new clocks and constraints to prevent the faulty TDT. The approach in [25] analyzes the violation of an existential property that ensures a location is reachable. A repair in their context relaxes time constraints such that extra behavior reaches a location as expected by the existential property. A repair in the our approach restricts time constraints.

2.3 Conclusion

Plenty of research exists that causally analyzes system failures. The concurrent interaction of several system components results in a complex behavior where some behaviors may lead to failures. In a complex interaction, rarely a single action is the cause of a failure, but rather, the interleaving of several actions causes a failure. Many of the referenced approaches analyze a single counterexample, and only few works aim at computing every cause for a system failure, but these approaches are mostly not complete. A complete analysis that scales for system models of realistic size is missing.

In real-time models not only the interleaving of actions, but also the time delays inside of a location can cause a failure. No approach exists to the best of our knowledge that causally analyzes the timing inside of a TDT to explain the occurrence of a failure. Causes for real-time failures can help, for instance, to choose correct time constraints in a real-time model, and in this way support design space exploration.

In this thesis, we present algorithms that compute causes based on discrete action orders, continuous time delays and time constraints in a system that potentially lead to a system failure. A full enumeration of causes helps to understand occurring failures and raises the confidence in the dependability of a system.

Chapter 3

Foundations

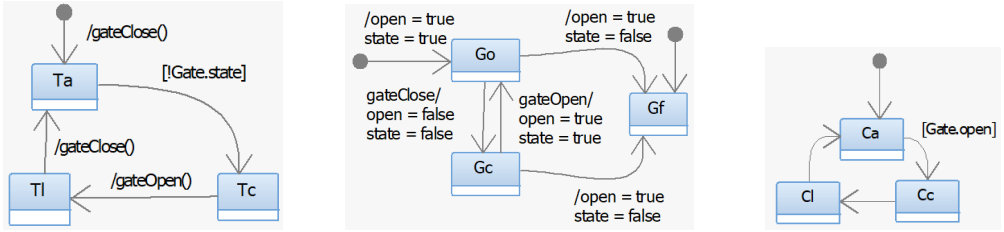
Contents

We present the foundation in this chapter. First, we introduce the system of a railroad crossing that we will use throughout the thesis to exemplify analyses of untimed systems. Causality Checking utilizes an explicit-state model checker to analyze a system model. The model checker searches the system model, which is given as a transition system, to find counterexamples that violate a given linear time property [110]. We also present the formal definition of actual causality in [62] and the cause definition used in the original description of Causality Checking as described in [102,104]. We describe the necessary foundations to analyze a real-time model in Section 3.6.

3.1 Running Example: Railroad Crossing

We exemplify our analyses for untimed systems, on the example of a railroad crossing.

Example 1 (Railroad)**.** *The railroad crossing consists of three components: a train, a car and a gate. The SysML state machines (STM) of the components are given in Figure 3.1. In the STMs, the locations are named with the actions that can occur in the railroad crossing. The STM transitions use the Boolean variables Gate.open that signals towards the car whether the gate is open, and the variable Gate.state that signals towards the train whether the gate is closed. An STM transition can be labeled with a pattern of the form event[guard]/effect. A guard is, for instance, Gate.open and has to evaluate to true when a transition is taken. An effect can assign a new value to a variable, or can send the messages denoted with (). A message is asynchronously transferred and triggers a transition that has the message as an event. In the railroad model, a train approaches a crossing (Ta) and sends a message* `gateClose` *to the gate, when the gate is closed then the train enters the crossing (Tc). Afterwards, the train can leave the crossing (Tl) and also sends a*



(a) STM `Train` (b) STM `Gate` (c) STM `Car`

Figure 3.1: STM Diagrams of Railroad Example

message ***gateOpen*** *to the gate. A car approaches the crossing (Ca). When the gate is open, the car can enter the crossing (Cc), and, afterwards, leave the crossing (Cl). The gate controller starts with an open gate. When receiving a message* ***gateClose****, the gate closes (Gc) and when receiving an message* ***gateOpen****, the gate opens (Go). At any time the gate can fail (Gf), which means that the gate is open while the train assumes that the gate is closed. A safety hazard occurs in this system when the train and the car are in the crossing at the same time, since this means a fatal accident. We, therefore, state the property that such a state shall not be reachable.*

3.2 Verification Formalisms of Transition Systems

A model checker interprets the STMs of a model as program graphs [14], interleaves and unfolds these program graphs to a transition system (TS), and searches in the TS for states that violate a desired property.

System Model. In order to interleave the STMs of a model, the current state of every STM is stored in a state vector together with the value of every variable in the model. The initial state vector of the railroad example is $(\mathrm{Ta}, \mathrm{Go}, \mathrm{Ca}, \mathit{Gate.open} = \mathit{true}, \mathit{Gate.state} = \mathit{true})$. Every assignment of the state vector is a state in a TS. A transition in an STM can change the state vector, and hence, is also a transition labeled with the name of the reached state in the TS.

Definition 1 (Transition System (TS) [14])**.** *A* transition system *is a tuple* $(S, Act, \rightarrow, I, AP, L)$ *where*

- S *is a finite set of states,*
- Act *is a finite set of actions,*
- $\rightarrow \subseteq S \times Act \times S$ *is a transition relation,*
- $I \subseteq S$ *is a set of initial states,*
- AP *is a set of atomic propositions, and*
- $L : S \rightarrow 2^{AP}$ *is a labeling function.*

A TS is deterministic iff $|I| \leq 1$ and for every state $s \in S$ and every action $a \in Act$ exists at most one relation $(s, a, s') \in\!\!\rightarrow$ with $s' \in S$ [14]. An *execution* τ of a TS is a possibly infinite alternating sequence $s_0, a_0, s_1, a_1 \ldots$ of states and actions that starts in a initial state and for $i \geq 0$, any triple (s_i, a_i, s_{i+1}), called a transition, is an element in $\rightarrow$. We denote that τ is an execution of a TS T by $\tau \vDash T$. An execution $s_0, a_1 \ldots s_n$ is *simple* iff $s_i \neq s_j$ holds for any two indices $i, j \in [0, n]$ with $i \neq j$. An *action trace* $\langle a_0, a_1, \ldots \rangle$ is the projection of an execution $s_0, a_0, s_1, a_1 \ldots$ on Act. An action trace is *simple* iff it is a projection of a simple execution. The set of all action traces of a TS T is the language $\mathcal{L}_T$ of T. An action trace $\sigma' = \langle a'_0, \ldots, a'_n \rangle$ contains another action trace $\sigma = \langle a_0, \ldots, a_m \rangle$ when σ is a non-contiguous subtrace of σ', written as $\sigma \sqsubseteq \sigma'$. Formally, $\sigma \sqsubseteq \sigma'$ holds iff the word $a'_0 \ldots a'_n$ is contained in the language obtained from the $\Sigma^* a_0 \Sigma^* a_1 \Sigma^* \ldots \Sigma^* a_m \Sigma^*$, where $\Sigma = \{a'_0 \ldots a'_n\}$. We write $\sigma \sqsubset \sigma'$ iff $\sigma \sqsubseteq \sigma'$ and $\sigma \neq \sigma'$. We refer to σ also as a subtrace of σ'. Let $\tau = s_0, a_0 \ldots a_m, s_{m+1}$ and $\tau' = s'_0, a'_0 \ldots a'_n, s'_{n+1}$ be executions from which traces σ and σ', respectively, have been derived by projection. We say τ' contains τ iff $\sigma \sqsubseteq \sigma'$. The $\sqsubseteq$ relation is transitive.

An *action trace class* is a set of action traces C where every action trace σ in C has the same alphabet $\mathcal{A} \subseteq Act$ and every action of the alphabet occurs in σ exactly once. Thus, every action trace in an action trace class has the same length $n = |\mathcal{A}|$. An action trace σ'' of a TS can contain an action a several times. In this case, we substitute every occurrence a in σ'' with a_i, where the index i is the number of occurrences of a up to the current action in σ'', add a_i to $\mathcal{A}$ and remove a.

Action traces can be space efficiently stored in a prefix tree [51] data structure. The edges in a prefix tree are labeled, in our context with actions. The path of some vertex is defined as a sequence of edges that leads from the considered vertex to the root vertex r. The path of any vertex in a prefix tree is simple and unique. An action trace of a path is the projection of the respective path on the set of actions. A path p contains another path p' iff the action trace of p contains the action trace of p'.

Linear Time Properties. A model checker searches in a transition system $T = (S, Act, \rightarrow, I, AP, L)$ for the violation of a property.

We denote the powerset over atomic AP as 2^{AP}. A *trace* ι is a sequence $A_0 A_1 \ldots$ with $A_i \in 2^{AP}$ for $i \geq 0$. A *linear-time property (LT property)* describes traces over 2^{AP} that satisfy the property.

Causality Checking analyzes LT properties that are given in *linear temporal logic (LTL)*. The syntax of LTL in presented in Definition 2. An LTL formula is formed over an atomic proposition set AP.

Definition 2 (Syntax of LTL [14])**.** *An LTL formula φ over the set AP of atomic propositions is formed according to the following grammar:*

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \circ\varphi \mid \varphi_1 U \varphi_2$$

where $a \in AP$.

We also use the *eventually* operator $\Diamond\varphi ::= \text{true}\ U\varphi$ and the *always* operator $\Box\varphi ::= \neg \Diamond \neg\varphi$.

The semantics of LTL is given in Definition 3. An LTL formula φ describes a set $Words(\varphi)$ of words over 2^{AP}. $Words(\varphi)$ corresponds to a subset of $(2^{AP})^{\omega}$.

Definition 3 (Semantics of LTL [14])**.** *Let φ be an* LTL formula *over AP. The LT property induced by φ is*

$$Words(\varphi) = \{\iota \in (2^{AP})^{\omega} \mid \iota \vDash \varphi\}$$

where the satisfaction relation $\vDash \subseteq (2^{AP})^{\omega} \times LTL$ is the smallest relation with the properties:

$\iota \vDash true$

$\iota \vDash a$ *iff* $a \in A_0$

$\iota \vDash \varphi_1 \wedge \varphi_2$ *iff* $\iota \vDash \varphi_1$ *and* $\iota \vDash \varphi_2$

$\iota \vDash \neg\varphi$ *iff* $\iota \nvDash \varphi$

$\iota \vDash \circ\varphi$ *iff* $\iota[1\ldots] = A_1A_2A_3\ldots \vDash \varphi$

$\iota \vDash \varphi_1 U \varphi_2$ *iff* $\exists j \geq 0.\iota[j\ldots] \vDash \varphi_2$ *and* $\iota[i\ldots] \vDash \varphi_1, \forall 0 \leq i < j.$

where $\iota = A_0A_1\ldots$ with $A_i \subseteq AP$ for $i \geq 0$.

An execution $s_0a_0s_1\ldots$ of a TS T contains states that can be mapped by L to a sequence $L(s_0)L(s_1)\ldots$ of atomic propositions. T satisfies an LTL formula φ when the atomic proposition sequence of every execution is in φ. Otherwise, an execution τ exists that witnesses the violation of φ. τ is named a counterexample and we write $\tau \nvDash \varphi$. We combine all counterexamples in a system T to a set Σ_B of "bad" executions [102]. The action trace σ_b derived from a counterexample τ_b with $\tau_b \nvDash \varphi$ is a *bad trace*, and we write $\sigma_b \nvDash \varphi$. In case an execution τ_g satisfies φ, we call the action trace σ_g of τ_g a *good trace* and write $\sigma_g \vDash \varphi$. We say that a transition system T *deterministically* violates a property φ iff for every action trace σ of T either $\sigma \vDash \varphi$ or $\sigma \nvDash \varphi$ holds, but not both. Notice that in this definition of determinism refers to an action trace and not to the trace of a property. T can non-deterministically violate a property violation because given two executions with an equivalent action trace, one execution can violate φ while the other does not.

Safety properties describe a subset of LT properties where a counterexample is a finite execution $s_0, a_0, s_1 \ldots s_n$ that leads to a property violating state.

Definition 4 (Safety Properties [14])**.** *An LT property P_{safe} over AP is called a* safety property *if for all words $\iota \in (2^{AP})^{\omega} \backslash P_{safe}$ there exists a finite prefix $\hat{\iota}$ of ι such that $P_{safe} \cap \{\iota' \in (2^{AP})^{\omega} \mid \hat{\iota}$ is a finite prefix of $\iota'\} = \varnothing$.*

Any such finite word $\hat{\iota}$ is called a bad prefix *for P_{safe}. The set of all bad prefixes for P_{safe} is denoted by $BadPref(P_{safe})$.*

For instance, the safety property "the train and the car are never in the crossing at the same time" of the railroad in Example 1 is a safety property. A finite counterexample exists with the bad action trace ⟨Ca, Cc, Ta, Gc, Tc⟩ that results in a hazard state where the car and the train have an accident in the crossing.

A safety property is regular when its set $BadPref(P_{\text{safe}})$ of bad prefixes is a regular language over 2^{AP} [14]. The correctness of regular safety properties in a system can be verified by reachability analysis [14].

Liveness properties are another subset of LT properties where, intuitively, in an execution something *good* has eventually to occur. A counterexample for such a property is an infinite execution where the good thing never occurs.

Definition 5 (Liveness Property [14])**.** *LT property P_{live} over AP is a* liveness property *whenever $pref(P_{live}) = (2^{AP})^*$.*

An arbitrary LT property P may be neither a safety nor a liveness property. However, P can always be described as the intersection of a liveness and a safety property [14].

State Space Exploration. In Causality Checking, we analyze regular safety properties. In order to find a violation of a regular safety property, the state space of a transition system can be searched by a depth-first search or breath-first search algorithm [14].

During a state space exploration, a state s may be reached by more than one execution: first by one execution and afterwards by another execution. In this case, we call s a *duplicate state*. A finite execution $s_0 a_0 s_1 \dots s_n$ of a TS where the last state s_n is a duplicate state is called a *duplicate execution*.

3.3 Strict Partial Orders

We use a *strict partial order* to express the action order in an action trace class. A strict partial order is based on the formalism of a *partial order*.

Definition 6 (Partial Order)**.** *A relation $\preceq \subseteq A \times A$ is called a* partial order *over a set A if, and only if,*

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

We call a constraint of the form $a \preceq b$ a *partial order constraint*. A partial order $\preceq$ is a *subset of* another partial order $\preceq'$, denoted by $\preceq \subseteq \preceq'$ if $\mathcal{A}_{\preceq} \subseteq \mathcal{A}_{\preceq'}$, and for every constraint $(a, b) \in \preceq$ it holds that $(a, b) \in \preceq'$. $\preceq$ is a true subset of $\preceq'$, written $\preceq \subset \preceq'$, if $\preceq \subseteq \preceq'$ and $\preceq \neq \preceq'$. A *strict partial order* distinguishes from a partial order in the property that it is *irreflexive*, which means that $\forall a \in A. a \not\preceq a$. We use the sign $\prec$ to denote a strict partial order.

A *transitive closure* of a binary relation R on a set X is the smallest relation on X that contains R and is transitive. We denote the transitive closure of $\prec$ by $\prec^+$. A (strict) partial order $\prec$ over set A is called (strict) total order if for any two elements a and b in A with $a \neq b$, either $a \prec b$ or $b \prec a$ is in $\prec^+$. An action trace $\sigma = \langle a_0 \dots a_n \rangle$ describes a total order $\prec_\sigma = \{a_0 \prec a_1, \dots, a_{n-1} \prec a_n\}$. The actions in σ are ordered according to a strict partial order $\prec'$ iff $\prec' \subseteq \prec_\sigma$. We also say that σ is *consistent* with the strict partial order $\prec'$, which we denote by $\sigma \vDash \prec'$, iff $\prec' \subseteq \prec_\sigma^+$ holds.

The function $i_\sigma(a)$ returns the index of an action a in an action trace σ of an action trace class. For an action trace class C, we say that an action b depends on a when in every action trace $\sigma \in C$ the index of a is smaller than the index of b. Formally, *a depends on b* if $\forall \sigma \in C. i_\sigma(a) < i_\sigma(b)$ holds. We express dependencies by a strict partial order. A dependency of action a on action b in an action trace class C is denoted by $a \prec_C b$. When C is clear from the context, we write $a \prec b$. In case $a \prec b$ holds, we say that a is a precondition for b. When neither $a \prec b$ nor $b \prec a$ holds, we say a and b are independent.

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

A partial order can be depicted by a directed graph. A *directed graph* G is a pair (V, E) of a finite nonempty set of vertices V and a set of edges $E \subseteq V \times V$.A *walk* of G is a finite sequence of vertices $v_0, v_1, \dots, v_n$ where for $0 \leq i \leq n$, $v_i \in V$ and for $0 \leq i \leq n-1$, $(v_i, v_{i+1}) \in E$. A vertex v is connected in G to a vertex u when a walk $v \dots u$ exists. A walk $v_0 \dots v_n$ is *unique* when removing any edge of the walk from G results in v_0 to be disconnected from v_n. A cycle is a walk $v_0 \dots v_k$ with $v_0 = v_k$ and $k \geq 1$. A *directed acyclic graph* (DAG) is a cycle free directed graph [34]. A (rooted) tree is a directed acyclic graph in which one vertex $r \in V$ is connected to every vertex $v \in V$ by a unique walk $r \dots v$. The reflexive transitive closure (V, E^*) of a directed graph (V, E) contains an edge (v, u) in E^* for every walk $v \dots u$ in (V, E).

3.4 Actual Causality

We now exemplify and introduce the formal definition of actual causality [59], and apply it to transition systems. This approach is called Causality Checking.

The actual cause definition uses structural equations as a model. A *structural equation model* (SEM) S is a tuple (U, V, R) with exogenous variables U, endogenous variables V and a relation R that describes the possible value range of the variables in $U \cup V$. R is nonempty and contains a finite set $R(Y)$ of possible values for any variable $Y \in U \cup V$. A assignment of exogenous variables describes a *world*. A *causal model* is a tuple (S, F) that describes a variable assignment by a SEM S and F assigns to every variable $v \in V$, a function F_v with the parameters $U \cup V / \{v\}$ and assigns a value in $R(v)$ to v. F is convergent, thus, recursively applying the functions in F on an assignment of V results in a stable assignment of V.

We can model the rock-throwing example from above with Boolean variables. Susi is throwing a rock S_T, the rock of Susi hits the bottle S_H, Bob is throwing as rock B_T, the rock of Bob hits the bottle B_H and the bottle shatters D. F assigns the following functions $F_{S_H}(S_T, B_T, B_H, D) \leftarrow S_T$, $F_{B_H}(S_T, B_T, S_H, D) \leftarrow B_T \wedge \neg S_H$ and $F_D(S_T, B_T, S_H, B_H) \leftarrow S_H \vee B_H$. The variables evaluate to 1 when the described event happens in a world. $D = 1$ is the effect that we want to explain. The world where Susi and Bob throw a rock ($S_T \leftarrow 1$, $B_T \leftarrow 1$) results by applying the functions in F repeatedly in a value assignment $S_T \leftarrow 1$, $S_H \leftarrow 1$, $B_T \leftarrow 1$, $B_H \leftarrow 0$ and $D \leftarrow 1$.

For this world, Susi and Bob throwing a rock are potential *actual causes* for shattering the bottle. In [59], contingency was introduced to block Bob's throw such that Susi's throw satisfies the counterfactual argument. Let us assume that Susi throwing the rock is the actual cause, formally, $\{S_T \leftarrow 1\}$. In the given world, the actual cause $\{S_T \leftarrow 1\}$ and the effect $D = 1$ applies. In order to check the counterfactual argument, we negate the cause $S_T \leftarrow 0$ and we block Bob by keeping the assignment $B_H \leftarrow 0$. The other variables evaluate to $S_H \leftarrow 0$, $B_T \leftarrow 1$ and $D \leftarrow 0$. The cause and the effect do not occur in this alternate world. Now, we do the same test for Bob's throw, and create an alternate world where he is not throwing a rock. In this alternate world holds $B_H \leftarrow 0$ but the effect $D = 1$ still applies because of Susi throwing a rock. Notice that we can also not prevent $D = 1$ by keeping a variable assignment, for instance, $S_H \leftarrow 1$ of the original world in this alternate world. Hence, Bob throwing a rock cannot be the actual cause, and Susi throwing a rock is the actual cause for shattering the bottle in the given world.

A formal definition of an actual cause $\vec{X} = \vec{x}$ with contingencies for an effect φ is given in Definition 7. The definition refers to a model M and a world $\vec{u}$ where φ and $\vec{X}$ evaluates to true (AC1). The actual cause for φ to occur is a subset of $\vec{u}$ where a modification of this assignment $\vec{X} = \vec{x'}$ results in a world where φ evaluates to false (AC2.1). In order to block other potential causes from inferring φ, we split the variables in $\vec{V}$ in a set $\vec{Z}$ and $\vec{W}$ where $\vec{Z} \cap \vec{W} = \varnothing$ and $\vec{X} \subseteq \vec{Z}$. We choose the set of variables W in a way that keeping the assignment $\vec{W} \leftarrow \vec{w}$ prevents other causes from inferring φ (AC2.1). Additionally, a different assignment to any subset of $\vec{W}$ or $\vec{Z}/\vec{X}$ is not allowed to prevent φ (AC2.2). By AC3 $\vec{X}$ satisfying AC1 and AC2 has to be minimal.

Definition 7 (Actual cause [59, 61])**.** $\vec{X} = \vec{x}$ *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.*

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. $(M, \vec{u}) \models_{SM} [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}]\neg\varphi$.
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* $\vec{W}'$ *of* $\vec{W}$ *and forall all subsets* $\vec{Z}'$ *of* $\vec{Z}/\vec{X}$.

AC3: $\vec{X}$ *is minimal, in the sense that no subset of* $\vec{X}$ *satisfies conditions AC1 and AC2.*

The rock throwing example satisfied Definition 7 with $\vec{X} = \{S_T \leftarrow 1\}$ and $\vec{W} = \{B_H \leftarrow 0\}$. In an alternative model without the variables S_H and B_H, the throws of Susi and Bob cannot be distinguished. For this alternate world, the cause is $\{S_T \leftarrow 1, B_T \leftarrow 1\}$. As we see, a cause also depends on the precision of the model.

3.5 Causality Checking

In Causality Checking, actual causality is applied to TSs in [102,104]. In a TS, the order of the actions can be important for the effect to occur. Therefore, the authors use event-order-logic (EOL) [104] to describe the action orders in a cause.

Event Order Logic (EOL) [102,105]. EOL is a logic that describes action orders for an action set and whether an execution is consistent with this action order. We say an execution τ of a TS satisfies an EOL formula ψ, denoted by $\tau \vDash_e \psi$, whenever τ contains the actions in the action set non-continuously in a sequence consistent with the order defined by ψ.

In more detail, ψ contains of a set $\mathcal{A}$ of *event variables* where every event variable in $\alpha_a \in \mathcal{A}$ is of type Boolean and according to Definition 8 evaluates to true when an action a occurs in τ. The function $val_A(\tau)$ returns the valuation of every action variable in $\mathcal{A}$ for τ.

Definition 8 (Valuation of the Set of Event Variables [102,105])**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, $\tau = s_0, a_1, s_1, a_2, \ldots a_n, s_n$ a finite execution of T and $\mathcal{A}$ the set of event variables then we define the function $val_{\mathcal{A}}(\sigma)$ as follows:*

$$val_{\mathcal{A}}(\tau) = (\alpha_{a_1}, \ldots, \alpha_{a_n}) \mid \alpha_{a_i} = \begin{cases} \text{true}, \textit{ if } \tau \vDash_e \alpha_{a_i} \\ \text{false}, \textit{ otherwise} \end{cases}$$

Further, we define $val_{\mathcal{A}}(\tau) = val_{\mathcal{A}}(\tau')$ if for all $\alpha_{a_i} \in \mathcal{A}$ the values assigned by $val_{\mathcal{A}}(\tau)$ and $val_{\mathcal{A}}(\tau')$ are equal and $val_{\mathcal{A}}(\tau) \neq val_{\mathcal{A}}(\tau')$ else.

An EOL formula describes the valid action order in an execution by connecting the event variables in ψ with different operators, as defined in Definition 9. A *simple EOL formula* does not infer an order of the described actions and uses only the Boolean connectives negation ($\neg$), conjunction $\wedge$ and disjunction ($\vee$). A *complex EOL formula* can express the order of action by the *ordered conjunction* operator $\owedge$. An EOL formula $a \owedge b$ with two event variables a and b is satisfied when a validates to true before b. The EOL logic also provides interval operators $\owedge_[$, $\owedge_]$ and $\owedge_< \phi \owedge_>$, where a simple EOL formula has to hold for an interval of an execution. For example, $\Psi \owedge_[\phi$ describes that after an execution satisfies Ψ, every action satisfies ϕ. The operator $\owedge$ binds stronger than $\wedge$, and $\neg$ binds stronger than $\owedge$.

Definition 9 (Syntax of Event Order Logic (EOL) [102,105])**.** *Simple EOL formulas over a set $\mathcal{A}$ of event variables are formed according to the following grammar:*

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

where $\alpha \in \mathcal{A}$ 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.

The semantics of every operator is formally given by the semantics of an EOL formula in Definition 10. For a more detailed description, we refer to [102, 105].

We adapt the indices in Formulae 3.12-3.15 of Definition 10 because of the following observations:

- The semantics of $\psi_1 \barwedge \psi_2$ is given in [102] as

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

  These semantics ignore the execution part $j..k$. For instance, we observe that the execution $\tau_1 = s_0, a, s_1, b, s_2$ wrongly satisfied the EOL formula $\neg a \barwedge \neg b$ for $j = 0$ and $k = 2$ because with $\psi_1 = s_0$ and $\psi_2 = s_2$, $s_0 \vDash \neg a$ and $s_2 \vDash \neg b$ hold. This satisfaction is wrong since b occurs after a in τ_1. Thus, we remove the index k in Definition 10 with the consequences that $\tau_1 \nvDash \neg a \barwedge \neg b$ correctly holds.

- We observe that the execution $\tau_2 = s_0, a, s_1, b, s_2, c, s_3$ does not satisfy $\psi_o = (a \barwedge b) \barwedge_{[} c$, following the semantics in [102], even though $\tau_2 \vDash \psi_o$ is intuitive interpretation. $\tau_2 \nvDash \psi_o$ holds because according to the semantics in [102], a subtrace $\tau[j..j]$ with a single event has to satisfy $a \barwedge b$ for any index $j \in [1, 3]$. It seems to be the result of a typo in the semantics definition in [102]. Thus, we alter $\tau[j..j] \vDash_e \psi$ in the semantics of EOL formulae $\psi \barwedge_{[} \phi$ and $\phi \barwedge_{[} \psi$ into $\tau[i..j] \vDash_e \psi$, and alter $\tau[j..j] \vDash_e \psi_1$ in $\psi_1 \barwedge_{<} \phi \barwedge_{>} \psi_2$ into $\tau[i..j] \vDash_e \psi_1$.

Definition 10 ((Adapted) Semantics of Event Order Logic (EOL) [102, 105])**.** *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 $\alpha_{a_i} \in \mathcal{A}$, over which ϕ, ϕ_1, ϕ_2 are built. Let $\tau = s_0, a_1, s_1, a_2, \ldots a_n, s_n$ a finite execution of T and $\tau[i..r] = s_i, a_{i+1}, s_{i+1}, a_{i+2}, \ldots a_r, s_r$ a partial execution. We define that an execution τ satisfies an event order logic formula ψ, written as $\tau \vDash_e \psi$, as follows:*

$$s_j \vDash_e \alpha_{a_i} \text{ iff } s_{j-1} \xrightarrow{a_i} s_j \tag{3.1}$$

$$s_j \vDash_e \neg\phi \text{ iff not } s_j \vDash_e \phi \tag{3.2}$$

$$\tau[i..r] \vDash_e \phi \text{ iff } \exists j : i \leq j \leq r \,.\, s_j \vDash_e \phi \tag{3.3}$$

$$\tau[i..r] \vDash_e \neg\phi \text{ iff } \forall j : i \leq j \leq r \,.\, s_j \vDash_e \neg\phi \tag{3.4}$$

$$\tau \vDash_e \psi \text{ iff } \tau[0..n] \vDash_e \psi, \textit{ where } n \textit{ is the length of } \tau. \tag{3.5}$$

$$\tau[i..r] \vDash_e \phi_1 \wedge \phi_2 \text{ iff } \tau[i..r] \vDash_e \phi_1 \textit{ and } \tau[i..r] \vDash_e \phi_2 \tag{3.6}$$

$$\tau[i..r] \vDash_e \phi_1 \vee \phi_2 \text{ iff } \tau[i..r] \vDash_e \phi_1 \textit{ or } \tau[i..r] \vDash_e \phi_2 \tag{3.7}$$

$$\tau[i..r] \vDash_e \neg(\phi_1 \wedge \phi_2) \text{ iff } \tau[i..r] \vDash_e \neg\phi_1 \textit{ or } \tau[i..r] \vDash_e \neg\phi_2 \tag{3.8}$$

$$\tau[i..r] \vDash_e \neg(\phi_1 \vee \phi_2) \text{ iff } \tau[i..r] \vDash_e \neg\phi_1 \text{ and } \tau[i..r] \vDash_e \neg\phi_2 \tag{3.9}$$

$$\tau[i..r] \vDash_e \psi_1 \wedge \psi_2 \text{ iff } \tau[i..r] \vDash_e \psi_1 \text{ and } \tau[i..r] \vDash_e \psi_2 \tag{3.10}$$

$$\tau[i..r] \vDash_e \psi_1 \vee \psi_2 \text{ iff } \tau[i..r] \vDash_e \psi_1 \text{ or } \tau[i..r] \vDash_e \psi_2 \tag{3.11}$$

$$\tau[i..r] \vDash_e \psi_1 \wedge \psi_2 \text{ iff } \exists j : i \leq j \leq r \,.\, \tau[i..j] \vDash_e \psi_1 \text{ and } \tau[j{+}1..r] \vDash_e \psi_2 \tag{3.12}$$

$$\begin{aligned}\tau[i..r] \vDash_e \psi \wedge_{[} \phi \text{ iff } (&\exists j : i \leq j \leq r \,.\, \tau[i..j] \vDash_e \psi \\ &\text{and } (\forall k : j \leq k \leq r \,.\, \tau[k..k] \vDash_e \phi))\end{aligned} \tag{3.13}$$

$$\begin{aligned}\tau[i..r] \vDash_e \phi \wedge_{]} \psi \text{ iff } (&\exists j : i \leq j \leq r \,.\, \tau[j..r] \vDash_e \psi \\ &\text{and } (\forall k : 0 \leq k \leq j \,.\, \tau[k..k] \vDash_e \phi))\end{aligned} \tag{3.14}$$

$$\begin{aligned}\tau[i..r] \vDash_e \psi_1 \wedge_{<} \phi \wedge_{>} \psi_2 \text{ iff } (&\exists j,k : i \leq j < k \leq r \,.\, \tau[i..j] \vDash_e \psi_1 \text{ and } \tau[k..r] \vDash_e \psi_2 \\ &\text{and } (\forall l : j \leq l \leq k \,.\, \tau[l..l] \vDash_e \phi))\end{aligned} \tag{3.15}$$

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

The subset relation operators between two EOL formulae are given in Definition 11. For example, the EOL formula a has an event variable set $\{a\}$ and is a true subset of the EOL formula $a \wedge b$ since $\{a\} \subseteq \{a, b\}$ and $\{a\} \neq \{a, b\}$.

Definition 11 (EOL Formula Subset Relation Operators [102])**.** *Let ψ_1 and ψ_2 be EOL formulae, where ψ_1 is built over the set of event variables $\mathcal{A}_1$ and ψ_2 is built over the set of event variables $\mathcal{A}_2$. We define the subset relation operators $\subset$ and $\subseteq$ on EOL formulae as follows:*

- $\subseteq$*: $\psi_1 \subseteq \psi_2$ iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$.*
- $\subset$*: $\psi_1 \subset \psi_2$ iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and not $\mathcal{A}_1 = \mathcal{A}_2$.*

The Original Cause Definition in Causality Checking in [102, 104]. In [102, 104], Definition 7 of an actual cause is applied to transition systems. They defined a world of a transition system to be an execution. The conditions in Definition 12 now formalize when an EOL formula ψ is a cause for the violation of a property φ. At least one execution τ exists in T that satisfies ψ and violates the property (AC1). In order to satisfy the counterfactual argument, AC2.1 claims that at least one execution τ' exists in T that does not satisfy ψ and satisfies the property. AC2.2 claims that for every execution τ'' that satisfy ψ, additional events W in τ'' that are not in τ have no influence on the violation of φ. AC3 ensures that ψ is a minimal EOL formula satisfying AC1, AC2.1 and AC2.2.

Definition 12 (Cause for a Property Violation [102, 104].)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system, and τ, τ' and τ'' be some executions 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 $val_{\mathcal{A}}(\tau)$ given in Definition 8 to represent the valuation of all variables in $\mathcal{A}$ for a given τ.*

- *AC1: There exists an execution τ, for which both $\tau \vDash_e \psi$ and $\tau \nvDash \varphi$ hold.*
- *AC2.1: $\exists \tau'$ s.t. $\tau' \nvDash_e \psi \wedge (val_Z(\tau) \neq val_Z(\tau') \vee val_W(\tau) \neq val_W(\tau'))$ and $\tau' \vDash \varphi$. In words, there exists an execution τ' where the order and occurrence of events is different from execution τ and φ is not violated on τ'.*
- *AC2.2: $\forall \tau''$ with $\tau'' \vDash_e \psi \wedge (val_Z(\tau) = val_Z(\tau'') \wedge val_W(\tau) \neq val_W(\tau''))$ it holds that $\tau'' \nvDash \varphi$ for all subsets of W. In words, for all executions where the events in Z have the value defined by $val_Z(\tau)$ 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.*

For Example 1, a cause ψ^A given in [102] is $\mathrm{Ta} \barwedge \mathrm{Ca} \barwedge \mathrm{Gf} \barwedge \mathrm{Cc} \barwedge_< \neg\mathrm{Cl} \barwedge_> \mathrm{Tc}$. Cl is not allowed to occur after Cc occurred since in case the car leaves the crossing before the train enters the crossing the property violation does not occur, and AC2.2 is not satisfied. Non-occurrence of events is defined in Definition 13, and describe a set Q of event variables that are not allowed to occur in an execution. By considering non-occurrence, a cause can satisfy AC2.2. We call an event represented by an event variable in Q a *non-occurrence event.*

Definition 13 (Non-Occurrence of Events [102, 104].)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and τ and τ'' executions 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 $\alpha_a \in Q$ with $Q \subseteq W$ on execution τ 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 $\tau'' \vDash_e \psi \wedge val_Z(\tau) = val_Z(\tau'') \wedge val_Q(\tau) \neq val_Q(\tau'') \wedge val_{W \setminus Q}(\tau) = val_{W \setminus Q}(\tau'')$ and $\tau'' \nvDash \varphi$ holds. In addition we require that for no other set Q' that satisfies the conditions described above, $Q' \subseteq Q$.*

ψ^A represents just one order of the actions that result in a property violation. Also, other orders of the actions represent action traces that result in a property violation. Thus, some orders in the given cause are not causal for the property violation and can be removed from the cause. The order condition OC in Definition 14 removes the non-causal orders from a cause. The result is a cause that describes several action orders that result in a property violation. For example, a resulting cause given in [102] is $\mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_< \neg\mathrm{Cl} \barwedge_> \mathrm{Tc}$.

Definition 14 (Order Condition [102,104])**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ a transition system, and τ, τ' executions of T. Let Ψ an EOL formula over Z that holds for τ 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: $\tau \vDash_e \chi \wedge \exists \tau' \in \Sigma_B : \tau' = \tau \wedge \tau' \nvDash_e \chi \wedge \tau' \vDash_e \chi_\wedge$.

Theorem 1 states that Causality Checking based on Definition 12 is complete and is proven in [102].

Theorem 1 (Causality Checking is Complete [102])**.** *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 $\forall \sigma \in T.\sigma \not\models \varphi \Rightarrow \sigma \models_e \psi$.*

Cause Probability Computation [102, 105] For a cause computed by Causality Checking, QuantUM can compute the probability of the cause, as well as, the probability of the overall system failure [102, 105].

Therefore, QuantUM uses the stochastic model checker Prism [97]. In Prism, the probability computation of QuantUM uses model checking of continuous timed Markov chains (CTMC) [13]. In a CTMC, each transition has a probability rate λ. A probability rate follows a exponential distribution $P = e^{-\lambda t}$. P is the probability of a transition to leave the current and transit to the next state. Since a state can be left by several transitions, the probabilities of the transitions coexist until one transition is taken. The overall probability P_s to leave a state s is the minimum of all exponential distribution functions of the state and thereby the sum of all transition rates $P_s = e^{-\Sigma\lambda_i t}$ [13].

QuantUM converts the STM diagrams of a system model to the input language of Prism and then computes the probability of the overall system failure. Afterwards, for each cause, QuantUM generates a process that describes the order and the events of a cause. The process transitions are synchronized with the transitions of the Prism system model such that when the event appears in the system model, the transition with the corresponding event name is also taken in the cause process. QuantUM executes Prism for every cause to compute the probability of the cause.

3.6 Modeling an Specification Formalisms for Timed Systems

In Definition 15, which is adapted from [27], we represent a timed system by a timed automaton (TA). Locations L represent the control state locations in which a TA may reside. Time is represented by a set of clocks C and progresses continuously in a timed automata while it resides in a location. The possible clocks values are restricted by *clock constraints* that are either assigned as *location invariants* to a location, or as *guards* to a transition in Θ. Given a set of *clocks* C, we denote by $\mathcal{B}(C)$ the set of all *clock constraints* over C, which are conjunctions of *atomic clock constraints* of the form $c \sim n$, where $c \in C$, $\sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. For the remainder of this section, we fix a finite set of clocks C.

Definition 15 (Timed Automaton (TA) [27])**.** *A timed automaton T is a tuple $T = (L, l^0, \Sigma, \Theta, I)$ where*

- *L is a finite set of locations,*
- *$l^0 \in L$ is an initial location,*

- Σ *is a finite set of actions,*
- $\Theta \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$ *is the transition relation, and*
- $I : L \to \mathcal{B}(C)$ *denotes a labeling of locations with clock constraints, referred to as location invariants.*

For $\theta \in \Theta$ *with* $\theta = (l, g, a, r, l')$*, we refer to* g *as the* guard *of* θ *and to* r *as its* clock resets.

In Definition 16, the operational semantics of T is given by a timed transition system (TTS). A *clock valuation* u is a function $C \to \mathbb{R}_{\geq 0}$. For a clock constraint b, we write $u \vDash b$ iff b evaluates to true in u. There are two types of transitions. An *action transition* is in Θ and models the execution of an action whose guard is satisfied. These transitions are instantaneous and may reset the specified clocks. The passage of time in a location is modeled by *delay transitions*. Both types of transitions guarantee that location invariants are satisfied in the pre and post state.

Definition 16 (Timed Transition System (TTS) [29])**.** *For a timed automaton* $(L, l^0, \Sigma, \Theta, I)$*, a timed transition system is a tuple* $H = (S, s_0, \Sigma, \to)$ *where* $S = \{(l, u) \in L \times \mathcal{R}^C_{\geq 0} \mid u \vDash I(l)\}$*,* s_0 *is the initial state* (l^0, u_0) *such that* u_0 *maps all clocks to* 0*, and a transition* $(l, u) \xrightarrow{t} (l', u')$ *is in* $\to$ *iff*

- *(action transition)* $t = (l, g, a, r, l') \in \Theta$*,* $u \vDash I(l) \wedge g$*,* $u' \vDash I(l')$ *and for all clocks* $c \in C$*,* $u'(c) = 0$ *if* $c \in r$ *and* $u'(c) = u(c)$ *otherwise; or*
- *(delay transition)* $t \in \mathbb{R}_{\geq 0}$*,* $l = l'$*,* $u \vDash I(l)$*,* $u' \vDash I(l)$ *and* $u' = u + t$*.*

An urgent location is a location that has to be left immediately without taking a delay transition [29].

A *run* [27] of a TTS is a sequence of states and actions of the form $s_0 \xrightarrow{t_1} \xrightarrow{a_1} s_1 \xrightarrow{t_2} \xrightarrow{a_2} \dots$ with $s_i = (l_i, u_i)$ where s_0 is the initial state and every t_i is in $\mathbb{R}_{\geq 0}$ with $(l_i, u_i) \xrightarrow{t_i} (l_i, u_{i+1})$ in $\to$, and every a_i is in Σ with $(l_i, u_{i+1}) \xrightarrow{(l_i, g, a_i, r, l_{i+1})} (l_{i+1}, u_{i+1})$ in $\to$. A *timed trace* [27] is a sequence of timed actions $\xi = (t'_1, a_1), (t'_2, a_2), \dots$ that is generated by a run of a TTS associated with a TA, where $t'_i = \sum_{0 < j \leq i} t_j$. We capture families of timed traces that perform the same sequence of action transitions but differ in their delay transitions by the notion of a *symbolic timed trace*.

Definition 17 (Symbolic Timed Trace (STT))**.** *A symbolic timed trace (STT) of a TA* T *is a sequence of transitions* $\tau = \theta_0, \dots, \theta_{n-1}$*. A* realization *of* τ *is a sequence of delay values* $\delta_0, \dots, \delta_n$ *such that there exists states* $s_0, \dots, s_n, s_{n+1}$ *with* $s_i \xrightarrow{\delta_i} \xrightarrow{\theta_i} s_{i+1}$ *for all* $i \in [0, n)$ *and* $s_n \xrightarrow{\delta_n} s_{n+1}$*. We say that a STT is* feasible *if it has at least one realization.*

The timed language for a TA T is the set of all its symbolic timed traces, which we denote by $\mathcal{L}_T(T)$. For an timed trace, the *untimed trace* is obtained by projection on T's set of actions Σ. The untimed language of T consists of words over

Σ, so that there exists at least one timed trace of T forming this word. Formally, for a timed trace $\xi = (t_1, a_1), (t_2, a_2) \dots$, the untime operator $\mu(\xi)$ returns an untimed trace $\xi_\mu = a_1 a_2 \dots$. We define the untimed language $\mathcal{L}_\mu(T)$ of the TA T as $\mathcal{L}_\mu(T) = \{\mu(\xi) \mid \xi \in \mathcal{L}_T(T)\}$. We represent a finite untimed language using a *Nondeterministic Finite Automaton* and an infinite untimed language using a *Nondeterministic Büchi Automaton*.

Definition 18 (Nondetermistic Finite Automaton (NFA) [29])**.** *A nondeterministic finite automaton is a tuple $M = (S, \Sigma, \rightarrow, S_0, F)$ where S denotes a finite set of states, Σ denotes an alphabet, $\rightarrow \subseteq S \times \Sigma \times S$ denotes a transition relation, $S_0 \subseteq S$ denotes the set of initial states, and $F \subseteq S$ denotes the set of acceptance states. We write $s \xrightarrow{a} s'$ when $(s, a, s') \in \rightarrow$. An execution of M is a sequence $s_0 \xrightarrow{a_0} s_1 \dots \xrightarrow{a_{n-1}} s_n$ such that $s_0 \in S_0$ and $s_n \in F$. A run $s_0, s_1, \dots, s_n$ of M is the projection of an execution $s_0 \xrightarrow{a_0} s_1 \dots \xrightarrow{a_{n-1}} s_n$ on the state sequence. M accepts a word $w = a_0, a_1, \dots, a_{n-1}$ when an execution $s_0 \xrightarrow{a_0} s_1 \dots \xrightarrow{a_{n-1}} s_n$ exists. The language $\mathcal{L}_f(M)$ recognized by M is the set of (finite length) words accepted by M.*

Definition 19 (Nondeterministic Büchi Automaton (NBA) [29])**.** *An NFA $B = (S, \Sigma, \rightarrow, S_0, F)$ is called a nondeterministic Büchi automaton in case it is used with an acceptance condition for infinite input sequences. For an infinite sequence $r = s_0 s_1 \dots$ of states, let $\inf(r) = \{s \in S \mid s = s_i \text{ for infinitely many } i\}$. An NBA B accepts an infinite word $w = a_1 a_2 \dots$, $w \in \Sigma^\omega$, iff B has an infinite run $r = s_0 s_1 \dots$ of states on w such that $s_0 \in S_0$, $inf(r) \cap F \neq \varnothing$ and $\forall i \in \mathbb{N}_0^+, (s_i, a_{i+1}, s_{i+1}) \in \rightarrow$. The language $\mathcal{L}(B) \subseteq \Sigma^\omega$ recognized by an NBA B is the set of infinite words accepted by B. $\text{pref}(\mathcal{L}(B))$ denotes the set of all finite prefixes of words in $\mathcal{L}(B)$.*

For a given NFA or NBA M, the *closure* $cl(M)$ denotes the automaton obtained from M by turning every state into an accepting state. We call M closed iff $M = cl(M)$. Notice that an NBA accepts a safety language if and only if it is closed [4].

We use *zones* as given in Definition 20 in order to abstract an infinite-state TTS into a finite-state *Zone Automaton*, c.f. Definition 21.

Definition 20 (Zone [29])**.** *A diagonal constraint is an extended clock constraint of the form $x - y \sim n$ with two clocks x and y. For a finite set C of clocks, let $[\![\varphi]\!]_C = \{u \in \mathbb{R}_{\geq 0}^C \mid u \vDash \varphi\}$ denote the set of clock evaluations u satisfying a conjunction φ of diagonal clock constraints. A subset $z \subseteq \mathbb{R}_{\geq 0}^C$ is called a zone if there exists φ for which $z = [\![\varphi]\!]_C$.*

Definition 21 (Zone Automaton (adapted from [29]))**.** *Assume a timed automaton $T = (L, l^0, \Sigma, \Theta, I)$. We define the zone automaton $[\![T]\!]_Z = (S_Z, s_Z^0, \Sigma_Z, \Theta_Z)$ for T such that $S_Z = \{(l, z) \mid l \in L, z \subseteq \mathbb{R}_{\geq 0}^C \text{ is a zone}\}$, $s_Z^0 = (l^0, \{u_0\})$, $\Sigma_Z = \Sigma \cup \{\delta\}$ and a transition relation $\Theta_Z \subseteq S_Z \times \Sigma_Z \times S_Z$. Let $z^\uparrow = \{u + d \mid u \in z, d \in \mathbb{R}_0^+\}$, and let $z[r]$ denote the clock reset for a clock set r in a zone z such that in the resulting zone every clock in r evaluates to 0. We write $l \overset{a}{\rightsquigarrow} l'$ for a transition $(l, a, l') \in \Theta_Z$. The transition relation Θ_Z is the smallest relation that satisfies the following rules:*

1. *If $(l, z) \in S_Z$, then $(l, z) \overset{\delta}{\rightsquigarrow} (l, z^\uparrow \cap [\![I(l)]\!]_C)$.*

2. If $l \xrightarrow{\varphi,a,r} l' \in \Theta$*, then* $(l,z) \overset{a}{\rightsquigarrow} (l', (z \cap [\![\varphi]\!]_C)[r] \cap [\![I(l')]\!]_C)$.

A trace $a_0 \dots a_n$ *is a trace of* Θ_Z *iff for* $0 \le i \le n$, $l_i \overset{a_i}{\rightsquigarrow} l_{i+1}$ *exists and* $l_0 = l^0$. *We call a trace of* Θ_Z *a symbolic trace.*

$\mathcal{L}_T([\![T]\!]_Z)$ *denotes the set of timed traces with time delays that satisfy the zones of at least one symbolic trace of* $[\![T]\!]_Z$.

Property Specification. We focus on the analysis of timed safety properties, which we characterize by an invariant formula that has to hold for all reachable states of a TA. These properties ensure, for instance, that in every location the value of a clock variable is not above, equal to or below a certain (integer) bound. Formally, let $T = (L, l^0, C, \Sigma, \Theta, I)$ be a TA. A *timed safety property* Π is a Boolean combination of atomic clock constraints and *location predicates* $@l$ where $l \in L$. A location predicate $@l$ holds in a state (l', u) of T iff $l' = l$. We say that an STT S witnesses a violation of Π in T if there exists a realization of S whose induced final state does not satisfy Π. We refer to such an STT as a *timed diagnostic trace* (TDT) of T for Π.

T satisfies Π iff all its reachable states satisfy Π. This problem can be decided using model checking tools such as Kronos [142] and UPPAAL [26]. UPPAAL in particular computes a finite abstraction of the state space of TAs using zones for the graph construction. Reachability analysis is then performed by an on-the-fly search of the zone graph. If the property is violated, the tool generates a feasible TDT that witnesses the violation. The implementation of the analyses in Part III analyzes TDTs generated by the UPPAAL tool, but we maintain that our results can be adapted to any other tool producing TDTs.

We further note that UPPAAL takes a *network of timed automata* (NTA) as input, which is a *Calculus of Communicating Systems* (CCS) [113] style parallel composition of timed automata $T_1 \mid \dots \mid T_n$, yielding a TA. Since our analysis and repair techniques focus on timing-related errors rather than synchronization errors, we use TAs rather than NTAs in our formalization. However, our repair analysis can be directly applied to the TA obtained by the parallel composition and the implementation in TarTar works directly on NTAs.

Part II

Causal Analysis of Untimed System Models

Chapter 4

Revised Cause Definitions for Untimed System Models

4.1 Introduction

The focus of Causality Checking presented in [102] is not to blame a single action in a transition system to be causal for a property violation but to explain the interplay of several actions leading to a property violation. A complete enumeration of causes allows us to know the ways a system can fail and creates trust in an implemented system. The cause definition underlying Causality Checking in [102] is Definition 12. We demonstrate by use of an example in Section 4.2 that Causality Checking based on Definition 12 is incomplete. We adapt the ideas on which Definition 12 is based on, and elaborate a revised cause definition to analyze the bad traces of a transition system that non-deterministically violate a property in Section 4.3 and a revised cause definition to analyze the bad traces of a transition system that deterministically violates a property in Section 4.4. The proofs show that Causality Checking based on the revised definitions is complete and sound.

4.2 Incompleteness of Original Cause Definition

We now show by use of an example that Causality Checking as presented in [102] is not complete, as stated by Theorem 1, that is quoted from [102], and proven in [102, Theorem 43]. For Example 1, the action traces $\sigma = \langle \mathrm{Ta}, \mathrm{Ca}, \mathrm{Go}, \mathrm{Gc}, \mathrm{Tc}, \mathrm{Gf}, \mathrm{Cc} \rangle$ and $\sigma' = \langle \mathrm{Ta}, \mathrm{Ca}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Cc} \rangle$ violate the given property. We will see that according to the definitions in [102], for σ, no cause ψ_σ with $\sigma \vDash \psi_\sigma$ exists. σ represents a behavior where the gate erroneously opens, and in σ' the gate falsely signals the train that the gate is closed. This difference between the behaviors is reflected in the order of Tc and Gf in σ and σ'. For σ, we create an EOL formula $\psi_\sigma = \mathrm{Ta} \barwedge \mathrm{Ca} \barwedge \mathrm{Go} \barwedge \mathrm{Gc} \barwedge \mathrm{Tc} \barwedge_< \neg\mathrm{Tl} \barwedge_> \mathrm{Gf} \barwedge_< \neg\mathrm{Tl} \barwedge_> \mathrm{Cc}$ and for σ', an EOL formula $\psi_{\sigma'} = \mathrm{Ta} \barwedge \mathrm{Ca} \barwedge \mathrm{Gf} \barwedge \mathrm{Tc} \barwedge_< \neg\mathrm{Tl} \barwedge_> \barwedge \mathrm{Cc}$. The minimality condition (AC3) in Definition 12 compares σ and σ' by a subset relation over the event variables in the corresponding EOL formulae. $\psi_{\sigma'} \subset \psi_\sigma$ holds by Definition 11 since $\{\mathrm{Ta}, \mathrm{Ca}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Tl}, \mathrm{Cc}\}$ is a subset of $\{\mathrm{Ta}, \mathrm{Ca}, \mathrm{Go}, \mathrm{Gc}, \mathrm{Tc}, \mathrm{Tl}, \mathrm{Gf}, \mathrm{Tl}, \mathrm{Cc}\}$. Because $\psi_{\sigma'} \subset \psi_\sigma$ holds, $\sigma \vDash_e \psi_{\sigma'}$ is falsely assumed in the proof of Theorem 1 in [102]. Instead $\sigma \nvDash_e \psi_{\sigma'}$ holds because of the order of Gf and Tc and, hence, neither $\psi_{\sigma'}$ nor ψ_σ is the expected cause for σ violating φ. Also, in Example 1, no subtrace of σ exists that violates the property. Hence, no cause ψ_σ according to Definition 12 for σ violating φ exists, and consequently, Theorem 1 does not hold. We will change the minimality condition for a cause from referring to a subset condition over events to a subtrace condition.

With this revision, a cause $\sigma \vDash_e \psi_\sigma$ exists, and completeness of the revised Causality Checking can be proven.

Furthermore, we state the following observations O_i that motivate additional modifications of the cause definition in [102].

O1 Following Definition 12, several causes presented in [102] are actually not causes according to Definition 12 because they are not minimal and, as a consequence, violate AC3. For instance, $\psi = \mathrm{Gf} \wedge ((\mathrm{Ta} \wedge (\mathrm{Ca} \barwedge \mathrm{Cc})) \barwedge_< \neg \mathrm{Cl} \barwedge_> \mathrm{Tc})$ presented in [102], is not a cause according to Definition 12 since $\psi' = \mathrm{Cc} \barwedge_< \neg \mathrm{Cl} \barwedge_> \mathrm{Tc}$ is a true subset of ψ and satisfies AC1-AC3 as shown in the proof below. A fundamental decision in [102] was that a cause should describe the causal process with every necessary event to result in a property violation. With this decision, ψ' should not be a cause and we will revise Definition 12 so that ψ is then a cause.

Proof. Assume TS and the property φ of the railroad example in Example 1. We now show that ψ' satisfies AC1-AC3 of Definition 12.

The action trace $\sigma = \langle \mathrm{Go}, \mathrm{Ca}, \mathrm{Cc}, \mathrm{Ta}, \mathrm{Gc}, \mathrm{Tc} \rangle$ satisfies ψ' and violates φ. Thus, σ witnesses that AC1 holds. The action trace $\sigma' = \langle \mathrm{Ca}, \mathrm{Cc}, \mathrm{Cl} \rangle$ satisfies φ and does not satisfy ψ' since σ' does not contain Tc. Hence, σ' is evidence that AC2.1 holds for ψ'. We ensure AC2.2 by contradiction and assume that an action trace σ'' exists such that $\sigma'' \vDash \psi'$ and $\sigma'' \vDash \varphi$. When $\sigma'' \vDash \psi'$ holds, the car has entered but not left the crossing and the train has also entered the crossing. Thus, the car and the train are in the crossing at the same time which violates φ, and contradicts the assumption that $\sigma'' \vDash \varphi$. Hence, σ'' cannot exist and AC2.2 holds. ψ' is also minimal (AC3) since removing any action Cc, Cl or Tc from ψ' allows an action trace that satisfies ψ' and φ. No subset of ψ' can satisfy AC2.2 and, thus, AC3 holds.

In summary, ψ' is a cause because AC1–AC3 holds for ψ'. □

O2 $\psi_{\sigma'}$ presented above satisfies Definition 12. It contains several order constraints, such as that Ta occurs before Ca, that are not causal since swapping these two events still results in a property violation. Even if the OC condition removes these non causal orders resulting in a cause $\psi'_{\sigma'}$, $\psi_{\sigma'}$ is still a cause satisfying Definition 12 that contains non causal order constraints. This casts doubt on whether Definition 12 really defines a cause. We integrate the OC condition into the revised cause definition such that only $\psi'_{\sigma'}$ is considered a cause.

O3 We now show that the OC condition in Definition 14 also creates causes that do not satisfy Definition 12. Assume a TS with an action trace $\langle a, c, b \rangle$ that satisfies a property and two action traces $\langle a, b, c \rangle$ and $\langle c, a, b \rangle$ violating the property. By using the OC condition, we obtain a cause $\psi_c = (a \barwedge b) \wedge c$. The action trace $\langle a, c, b \rangle$ satisfies ψ_c but also satisfies the property. This violates AC2.2, and ψ_c cannot be a cause according to Definition 12. Instead, $a \barwedge b \barwedge c$ and $c \barwedge a \barwedge b$ are the causes that satisfy Definition 12. Hence, a cause relaxed

by the OC condition may over-approximate and result in too many action traces satisfying the cause. By integrating the OC condition into the revised cause definition, we remove this over-approximation.

O4 Definition 12 uses a set W to describe event variables that do not affect the violation of φ. When the variables of W have no effect on the violation of a property then the events in W are not part of the cause since a cause is minimal (AC3). In Definition 7, W was used to distinguish between several causes that are simultaneously satisfied in a world to detect the actual cause which is responsible for a property violation. We cannot decide by Definition 12 which cause is the actual cause for a property violation, as the following example shows. The action trace $\langle$Go, Ca, Cc, Ta, Gc, Gf, Tc$\rangle$ violates the property in Example 1 and, for this violation $((\mathrm{Go} \wedge \mathrm{Gc}) \wedge (\mathrm{Ca} \wedge \mathrm{Cc}) \wedge \mathrm{Ta}) \wedge_{<} \neg\mathrm{Tl} \wedge_{>} \mathrm{Tc}$ and $(\mathrm{Gf} \wedge (\mathrm{Ca} \wedge \mathrm{Cc}) \wedge \mathrm{Ta}) \wedge_{<} \neg\mathrm{Tl} \wedge_{>} \mathrm{Tc}$ are causes according to Definition 12. Which of the two causes is the actual one, cannot be decided by Definition 12. The aim of Causality Checking is to compute every cause in a system and to determine the possible actual causes for a given bad trace. In the example above, we call both causes actual since the action trace satisfies both causes. For this understanding of an actual cause, W is not necessary in Definition 12. We can remove W from the definition which simplifies its formulation.

O5 AC2.2 in Definition 12 ensures the regularity property that whenever the cause occurs, the effect also occurs. Causality Checking based on Definition 12 analyzes transition systems with action traces that deterministically violate or satisfy a given property in order to ensure AC2.2 in Definition 12 by analyzing the non-occurrence of events.

As an example for a transitions system that non-deterministically violates a property, assume a TS T where an action trace $\langle a, b\rangle$ violates a property φ, and another action trace $\langle a, b\rangle$ satisfies φ. Causality Checking would compute the EOL formula $\psi = a \wedge b$ which does not satisfy AC2.2 of Definition 12 since $\langle a, b\rangle \vDash \psi$ and $\langle a, b\rangle \vDash \varphi$. Non-occurrence cannot deal with this action trace, since $\langle a, b\rangle$ satisfies φ and no non-occurrence event exists in $\langle a, b\rangle$ to explain the satisfaction of φ. For T, no cause according to Definition 12 exists. A realistic example for a system model that non-deterministically violates a given property is the autonomous driving example presented in Chapter 7. When an action trace violates and satisfies an property, some causal relation is not modeled in the underlying TS and, hence, the cause cannot be computed. For those TSs, we are interested to compute the causal relations of the cause that are modeled in the TS, and we call these relations a *weak cause*. We present a weak cause definition that does not ensure AC2.2 for transition systems that non-deterministically violate φ, and afterwards a cause definition that ensures AC2.2 for a TS that deterministically violate φ.

These observations motivate us to revise the cause definition underlying Causality Checking.

4.3 A Cause for Non-Deterministic Property Violations

We now derive the weak cause definition that adopts the underlying ideas of Definition 12 to analyze the bad traces of transition systems that do not deterministically violate a property.

In the new cause definitions presented below, we assume a world to be an action trace of a transition system T and the effect to be the violation of a regular safety property φ. An action trace determines a total ordering of the actions that it contains. For instance, $\sigma_1 = \langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Tc} \rangle$ is an action trace of Example 1, and violates the property that the car and the train are never in the crossing at the same time. σ_1 determines the total order $\{\mathrm{Ca} \prec \mathrm{Ta}, \mathrm{Ta} \prec \mathrm{Gf}, \mathrm{Gf} \prec \mathrm{Cc}, \mathrm{Cc} \prec \mathrm{Tc}\}$ for the action set $\{\mathrm{Ca}, \mathrm{Cc}, \mathrm{Ta}, \mathrm{Tc}, \mathrm{Gf}\}$. The total order of σ_1 also describes implicit transitive order constraints, for instance, $\mathrm{Gf} \prec \mathrm{Tc}$. We assume in Causality Checking that σ_1 violates the property because of some of the actions and their ordering determined by σ_1. Based on this assumption, a cause is a subset of the total order that σ_1 determines.

In the following, we determine the causal actions and causal order constraints entailed in σ_1 by the counterfactual argument. Since every subtrace of σ_1 either satisfies the property or is not an action trace of Example 1, every action in σ_1 is causal.

The action order of two actions in a bad trace satisfies the counterfactual argument when for a different action order the property violation would not have occurred. We now discuss which order relations in σ_1 are causal since they satisfy the counterfactual argument. For instance, the order constraint $\mathrm{Gf} \prec \mathrm{Tc}$ is causal since any action trace in Example 1 with $\mathrm{Tc} \prec \mathrm{Gf}$ of shorter or equal length as σ_1 either satisfies the property or is not an action trace of the considered TS. By the same argument, the other causal order constraints in σ_1 are $\mathrm{Ca} \prec \mathrm{Cc}$ and $\mathrm{Ta} \prec \mathrm{Tc}$. An order constraint that is not causal in σ_1 is, for instance, $\mathrm{Gf} \prec \mathrm{Cc}$ because swapping the order of Gf and Cc in σ_1 still results in an action trace that violates the property.

Whether the property is no longer violated after swapping two actions in an action trace is not the only reason for an order to be causal. For instance, the action trace $\sigma_2 = \langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gc}, \mathrm{Tc}, \mathrm{Gf}, \mathrm{Cc} \rangle$ violates the property in Example 1 and no subtrace of it violates the property. Changing the order of Tc and Gf in σ_2 results in an action trace $\sigma_2' = \langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gc}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Cc} \rangle$ that still violates the property. In σ_2, every event is causal whereas we can remove Gc from σ_2' and have found a subtrace $\sigma_2'' = \langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Cc} \rangle$ that still results in a property violation. Hence, the cause for the property violation by σ_2 cannot be the cause for the property violation by σ_2' since a smaller cause without Gc exists for the second violation (AC3 in Definition 7). Hence, σ_2 and σ_2' satisfy different causes, and the order constraint $\mathrm{Tc} \prec \mathrm{Gf}$ is causal in σ_2 since it separates these causes.

An order constraint in a cause describes that an action depends on the occurrence of another action in order to reach a property violation. The order relation in a cause is a dependency relation as defined in Section 3.3 that we describe by a strict partial order for the following reasons:

- An action can not depend on itself, otherwise, the action can never occur

(*irreflexivity*).

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

An action trace σ of a TS T can contain an action a several times. In this case, we substitute every occurrence a in σ with a_i where the index i is the number of occurrences of a up to the current occurrence of a in σ.

We now present the revised cause definition in Definition 22. A cause ϕ according to Definition 22 for the violation of a regular safety property φ is described by a strict partial order $\prec$ over $\mathcal{A}$ where $\mathcal{A}$ is the action set of minimal-length bad traces consistent with $\prec$. We call ϕ a *weak cause* when ϕ does not comply with AC2.2 of Definition 7 and instead can analyze the bad traces also of a transition system that non-deterministically violate or satisfy a property.

In line with AC1 of Definition 12, CC1 in Definition 22 ensures that a behavior of a system exists where the cause and the effect occur. In comparison to AC1, CC1 is more strict. It additionally ensures that any ordered sequence σ of the actions in $\mathcal{A}$ with $\sigma \vDash \prec$ is a bad trace of T and no subtrace of σ violates the property. We call every σ satisfying CC1 a *causal trace*. The intuition behind a causal trace is that the underlying execution is a minimal counterexample of T. CC1 ensures that a causal trace is minimal and does not contain a shorter bad trace. This restriction is on the one side a minimality condition on the events in $\mathcal{A}$, and adds on the other side order constraints to $\prec$ to ensure that no causal trace of $\prec$ satisfies another cause (cf. σ_2 and σ_2''). In case a causal trace satisfies another cause with less events, then $\prec$ is not the actual cause because of AC3 in Definition 12. As mentioned before, Tc $\prec$ Gf is an example for such a distinguishing order constraint. In addition, CC1 is an interpretation of the regularity argument that if an action trace is consistent with the cause and contains exactly the events in $\mathcal{A}$, then the effect occurs.

Condition CC2 identifies an action trace σ' in which either some events of $\mathcal{A}$ do not occur or their order differs from $\prec$, and which does not violate φ. Hence, σ' does not satisfy $\prec$ but φ, and witnesses the satisfaction of the counterfactual argument (AC2 in Definition 12).

The minimality condition CC3 is a subset relation as in Definition 12. In Definition 22, the subset relation is not defined over events but over the partial order constraints in $\prec$. Notice, that a subset relation over a partial order represents a subtrace relation. Thus, $\prec$ contains only the order constraints that are causal for the violation of φ.

Definition 22 (Weak Cause for a Regular Safety Property Violation)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A strict partial order $\prec$ with an event set $\mathcal{A} \subseteq Act$ is a* weak cause *ϕ for the violation of the regular safety property φ, if the following conditions are satisfied:*

CC1 *Any ordered sequence σ of $\mathcal{A}$ with $\sigma \vDash \prec$*

1. *is a bad trace of* T*, and*
2. *no true subtrace* $\sigma' \sqsubset \sigma$ *violates* φ*.*

CC2 *There exists an action trace* σ' *of* T*, such that* $\sigma' \not\models \prec$ *and* $\sigma' \models \varphi$*.*

CC3 $\prec$ *is minimal, i.e., no true subset of* $\prec$ *satisfies CC1 and CC2.*

Let ϕ be a cause according to Definition 22 given by a strict partial order $\prec$. We say an action trace σ satisfies ϕ denoted by $\sigma \models \phi$ iff σ is consistent with $\prec$.

The counterfactual test in CC2 prevents that a cause exists for a trivial violation where the initial state violates φ. If the initial state does not satisfy φ, then φ cannot be satisfied by any action trace of T. This implies that it is inevitable to violate φ, and hence, no cause based on a counterfactual argument should be identified.

Example 2. *We now show that* $\phi_r = \{Ca \prec Cc, Ta \prec Tc, Gf \prec Tc\}$ *is a cause according to Definition 22 with respect to Example 1 and the regular safety property* $\varphi := \square\neg(Cc \wedge Tc)$*. Therefore, we check that* ϕ_r *fulfills the conditions CC1, CC2 and CC3 of Definition 22. The action set* $\mathcal{A}_r$ *of* ϕ_r *is* $\{Ca, Cc, Ta, Tc, Gf\}$*.*

- *CC1 is fulfilled since the set of linear orders of* $\mathcal{A}_r$ *that are consistent with* ϕ_r *is identical to the following set of action traces violating* φ*:*

Ca Cc Ta Gf Tc,	*Ta Ca Gf Cc Tc,*	*Ca Gf Cc Ta Tc,*	*Ta Ca Gf Tc Cc,*
Ca Ta Cc Gf Tc,	*Gf Ca Ta Cc Tc,*	*Gf Ca Cc Ta Tc,*	*Ta Gf Ca Tc Cc,*
Ta Ca Cc Gf Tc,	*Ta Gf Ca Cc Tc,*	*Gf Ca Ta Tc Cc,*	*Gf Ta Ca Tc Cc,*
Ca Ta Gf Cc Tc,	*Gf Ta Ca Cc Tc,*	*Ca Gf Ta Tc Cc,*	*Ta Gf Tc Ca Cc,*
Ca Gf Ta Cc Tc,	*Ca Cc Gf Ta Tc,*	*Ca Ta Gf Tc Cc,*	*Gf Ta Tc Ca Cc*

  Notice, that all action traces in this set are minimal, i.e., they do not contain subtraces that violate φ*. We now show that there exist only* 20 *sequences of* $\mathcal{A}_r$ *satisfying* ϕ_r*. For the events* $\{Ta, Gf, Tc\}$ *only the order* Ta, Gf, Tc *and* Gf, Ta, Tc *are orders consistent with* ϕ_r*. In both orders, we can insert* Ca *at four different indices which leaves the higher indices in the action trace as possible index for* Cc *because of* $Ca \prec Cc$*. Hence, there are* $4 + 3 + 2 + 1 = 10$ *possibilities to insert* Ca *and* Cc *in* Ta, Gf, Tc *and again* 10 *possibilities for* Gf, Ta, Tc*. In summary, there are these* 20 *orderings of* $\mathcal{A}_r$ *that satisfy* ϕ_r*.*

- *CC2 is fulfilled since the action trace*

  $$\sigma' = \langle Ca, Ta, Gf, Tc \rangle$$

  with $\sigma' \not\models \phi_r$ *and* $\sigma' \models \varphi$ *is in* T*.*

- *We further need to check condition CC3 for* ϕ_r*.* ϕ_r *is not minimal when it contains either a superfluous event or a superfluous partial order constraint. In Example 1, no action trace exists that violates* φ *with fewer events than the event set in* ϕ_r*. Thus, no event can be removed without satisfying* φ*. Removing one of the partial order constraints* $Ca \prec Cc$*,* $Ta \prec Tc$ *and* $Gf \prec Tc$ *in* ϕ_r *would allow action orders of* $\mathcal{A}$ *that are consistent with* ϕ_r *but are not an action trace of* T *and, hence, CC1 would be violated. Thus,* ϕ_r *is minimal and CC3 is satisfied.*

A system model can contain several causes where every cause by itself describes action traces that violate a property. We combine these causes in a *cause set*.

Definition 23 (Cause Set)**.** *Assume a transition system T and a regular safety property φ. A* cause set *is a set $\Phi = \{\phi_1, \ldots, \phi_n\}$ of causes where every cause ϕ_i in Φ is satisfied by at least one action trace σ of T that violates φ. Formally, $\forall \phi_i \in \Phi . \exists \sigma \in \mathcal{L}_T \; . \; \sigma \vDash \phi_i \wedge \sigma \nvDash \varphi$.*

A cause set $\Phi := \{\phi_1, \ldots, \phi_n\}$ can be interpreted as a disjunction of causes where a bad trace σ_b satisfies Φ iff there exists a cause ϕ_i in Φ such that $\sigma_b \vDash \phi_i$ holds. Definition 24 defines a cause set Φ to be *complete* for a given TS T and a regular safety property φ when every bad trace of T is consistent with at least one cause in Φ.

Definition 24 (Completeness of Cause Set)**.** *For a TS T and a regular safety property φ, a cause set $\Phi := \{\phi_1, \ldots, \phi_n\}$ is* complete *iff every bad trace of T is consistent with a cause ϕ_i in Φ. Formally, $\forall \sigma \in \mathcal{L}_T \; . \; \sigma \nvDash \varphi \rightarrow \exists \phi_i \in \Phi \; . \; \sigma \vDash \phi_i$.*

Definition 24 defines a cause set Φ to be *sound* by Definition 25 when Φ contains only elements that satisfy a cause definition, and Φ does not contain any superfluous cause that is not satisfied by any bad trace of a transition system.

Definition 25 (Soundness of Cause Set)**.** *For a transition system T and a regular safety property φ, a cause set $\Phi = \{\phi_1, \ldots, \phi_n\}$ is* sound *iff every cause ϕ_i in Φ is a cause according to a cause definition and an action trace σ in $\mathcal{L}_T$ exists where $\sigma \nvDash \varphi$ and $\sigma \vDash \phi_i$ holds.*

We are interested in a complete and sound algorithmic approach called Causality Checking to compute a complete and sound cause set for a given system and a given regular safety property.

Causality Checking Based on Definition 22 computes a cause for each bad trace with a simple execution of a transition system and combines these causes to a cause set. Notice, that Causality Checking will terminate since a transition system contains a finite number of states, hence, there are only a finite number of simple executions.

Completeness and Soundness of Causality Checking Based on Definition 22. We now show that Causality Checking based on Definition 22 is complete and sound.

Theorem 2 states that Causality Checking is complete iff it returns a complete cause set for a given TS and a given regular safety property. Causality Checking can only be complete when, as stated by Lemma 1, for each possible bad trace in the TS under analysis, there exists a cause representing this trace. A transition system violates a property trivially, when an initial state violates the property. The bad trace of an initial state is an empty action trace. The corresponding cause is the empty partial order relation $\{\}$, but every action trace σ' of the underlying transition system is consistent with $\{\}$ and, hence, CC2 is unsatisfiable. Hence, no cause exists for a transition system that violates a property trivially. In the following, we only

consider TSs that do not violate a regular safety property trivially. This assumption ensures that the empty action trace fulfills CC2 for every bad trace in a TS. Lemma 1 states that a cause has to exists for any bad trace since either the total order of the bad trace is the cause or a subset of it satisfies CC1 to CC3.

Lemma 1 (Existence of Cause)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system where any state in I satisfies the regular safety property φ. For each action trace σ of T that violates φ, there exists a cause ϕ with $\sigma \vDash \phi$.*

Proof. Let σ be an action trace in a TS T that violates a regular safety property φ, and no initial state of T violates φ. We construct a cause ϕ with $\sigma \vDash \phi$ that satisfies CC1-CC3 according to Definition 22.

σ violates φ. In case σ contains subtraces that also violate φ, we select from σ a length-minimal subtrace σ'. Otherwise, we set $\sigma' = \sigma$. In both cases, $\sigma' \sqsubseteq \sigma$ holds which implies that σ is consistent with the total order $\prec_{\sigma'}$ of σ'. $\prec_{\sigma'}$ satisfies CC1 since $\prec_{\sigma'}$ describes by construction a minimal total order of a single action trace σ' of T that violates φ. $\prec_{\sigma'}$ also satisfies CC2 since the empty trace is a subtrace of σ' and reaches an initial state that satisfies φ by assumption. Yet, we proved that $\prec_{\sigma'}$ satisfies CC1 and CC2.

We now determine ϕ as a subset of $\prec_{\sigma'}$ where CC1–CC3 holds. Notice, that only a finite number of subsets of $\prec_{\sigma'}$ exists. When no subset of $\prec_{\sigma'}$ satisfies CC1 and CC2, then CC3 is also satisfied for $\prec_{\sigma'}$. In this case $\phi = \prec_{\sigma'}$. Otherwise, there is only a finite number of subsets, and we determine a minimal subset ϕ of $\prec_{\sigma'}$ that satisfies CC1 and CC2. In both cases, a ϕ with $\phi \subseteq \prec_{\sigma'}$ exists and $\sigma \vDash \phi$ holds by construction. Hence, ϕ is the searched cause for σ violating φ. □

With this lemma at hand, we can now prove the following theorem that Causality Checking is complete.

Theorem 2 (Causality Checking is Complete)**.** *Causality Checking based on Definition 22 is complete if it returns for a transition system T and a regular safety property φ, a complete cause set (see Definition 24).*

Proof. Causality Checking considers every bad trace σ of a simple execution of T and adds a cause ϕ_σ that exists by Lemma 1 to Φ. For these bad traces a cause is by construction in Φ, and it remains to be proven that every bad trace of T with an execution that contains a loop is consistent with a cause in Φ.

Assume a bad trace $\sigma_l = \langle a_1 \dots a_k \dots a_l \dots a_n \rangle$ of an execution $s_0 a_1 \dots s_i a_k \dots s_i a_l \dots a_n$ with a loop through state s_i, then there exists an execution $s_0 a_1 \dots s_i a_l \dots a_n$ with the bad trace $\sigma'_l = \langle a_0 \dots a_l \dots a_n \rangle$ without the loop. Since $\sigma'_l \sqsubset \sigma_l$ holds and by Lemma 1 a cause ϕ'_l with $\sigma'_l \vDash \phi'_l$ exists, also $\sigma_l \vDash \phi'_l$ holds. Hence, σ_l is consistent with a cause in Φ that Causality adds for σ'_l to Φ.
□

We now want to ensure that Causality Checking is sound. Theorem 3 states that Causality Checking is sound if it returns for a transition system and a regular safety property a sound cause set according to Definition 25.

In the proof of Theorem 3, we use Lemma 2, which states that for every partial order a sequence σ exists for which $\sigma \vDash \prec$. Since the proof of Lemma 2 uses only the antisymmetry and transitivity properties of a partial order, it also holds for strict partial orders.

Lemma 2. *For any partial order $\prec$ over a set $\mathcal{A}$, there exists an ordered sequence σ of $\mathcal{A}$ for which $\sigma \vDash \prec$ holds.*

Proof. Assume a partial order $\prec$ over a set $\mathcal{A}$ of size n. By induction over the elements in $\mathcal{A}$, we now construct an ordered sequence $\sigma = \langle a_1, \ldots, a_n \rangle$ with $\sigma \vDash \prec$. For $1 \leq i \leq n$, we ensure that $\sigma_i = \langle a_1, \ldots, a_i \rangle$ is consistent with a partial order $\prec_i$ that consists of every order constraint in $\prec$ of the form (a_j, a_i), with $1 \leq j < i$.

Base Case: Consider an empty sequence $\sigma_0 = \langle\rangle$ and an empty partial order relation $\prec_0 = \{\}$ over an empty event set $A_0 = \varnothing$. For σ_0 and $\prec_0$, $\sigma_0 \vDash \prec_0$ and $\prec_0 \subseteq \prec$ hold.

Induction Step: Assume $\sigma_{i-1} \vDash \prec_{i-1}$ and $\prec_{i-1} \subseteq \prec$ over event set $\mathcal{A}_{i-1}$. We choose an element a_i of $\mathcal{A}$ that is not yet in σ_{i-1} and for every $(a_j, a_i) \in \prec$, a_j is already in σ_i. a_i has to exist since otherwise a relation (a_j, a_i) exists in $\prec$ with a_j not in σ_i, and we can choose a_j as a_i. This different choice of a_i can occur only a finite number of times, since, by Definition 6, $\prec$ is antisymmetric, transitive and finite. These properties ensure that any two elements a and b in $\mathcal{A}$ with $a \neq b$ either satisfy $(a, b) \in \prec$, or $(b, a) \in \prec$, but not both. We construct $\sigma_i \coloneqq \langle a_0, \ldots, a_{i-1}, a_i \rangle$ by appending a_i to σ_{i-1}. In order to construct $\prec_i$, we add every order constraint in $\prec$ of the form (a_j, a_i) to $\prec_{i-1}$ and set $\mathcal{A}_i = \mathcal{A}_{i-1} \cup \{a_i\}$. By construction, $\sigma_i \vDash \prec_i$ and $\prec_i \subseteq \prec$.

After n induction steps, every element in $\mathcal{A}$ is in $\mathcal{A}_n$, hence, it holds that $\mathcal{A}_n = \mathcal{A}$, $\sigma_n \vDash \prec_n$ and $\prec_n = \prec$. Consequently, σ_n is the to be constructed σ for which $\sigma \vDash \prec$ holds. □

Theorem 3 (Causality Checking is Sound)**.** *Causality Checking based on Definition 22 is sound if it returns for a transition system T and a regular safety property φ, a sound cause set (see Definition 25).*

Proof. Assume that Causality Checking returns a cause set Φ for a TS T and a regular safety property φ. We now prove by contradiction that every cause $\prec_i$ in Φ is a cause according to Definition 22 and an action trace in T violating φ exists with $\sigma \vDash \prec_i$. Therefore, we assume a cause $\prec$ in Φ that is either not a cause according to Definition 22 or not a cause of T since no bad trace σ in T exists where $\sigma \vDash \prec$ holds.

In case $\prec$ is not a cause according to Definition 22, Causality Checking would not have added $\prec$ to Φ, as assumed. Thus, $\prec$ is a cause according to Definition 22, and the other case has to hold that no bad trace σ is consistent with $\prec$. By Lemma 2, an ordered sequence σ' of the elements in $\mathcal{A}$ has to exist such that $\sigma' \vDash \prec$ holds. Since σ' is an ordered sequence of the events in $\prec$ and $\sigma' \vDash \prec$, also $\sigma' \in \mathcal{T}$ and $\sigma' \nvDash \varphi$ holds by CC1. This σ' is the σ that we assumed not to exist.

In every case, the assumed $\prec$ cannot exist. Consequently, Causality Checking based on Definition 22 is sound.

□

4.4 A Cause for Deterministic Property Violations

We expect that a cause is regular, in other words, whenever the cause occurs, the effect also occurs. A weak cause does not ensure regularity because it applies also to TSs that non-deterministically violate a property as we have seen in observation O5 of Section 4.2. For TS that deterministically violate a given property, we now present a cause definition that ensures regularity. In this cause definition, we adopt the regularity argument AC2.2 of Definition 12 and the idea to ensure AC2.2 by analyzing non-occurrence.

In Example 2, ϕ_r is a weak cause according to Definition 22. ϕ_r is not regular since the car can still leave the crossing (Cl) before the train enters the crossing (Tc). For instance, the action trace

$$\hat{\sigma}_r = \langle \mathrm{Ta}, \mathrm{Ca}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Cl}, \mathrm{Tc} \rangle$$

satisfies ϕ_r, but does not violate φ. The occurrence of Cl in $\hat{\sigma}_r$ prevents the property violation. $\hat{\sigma}_r$ shows that the non-occurrence of event Cl in ϕ_r satisfies the counterfactual argument and is therefore causal for the violation of φ.

The occurrence of a non-occurrence event does not prevent a property violation in every action trace. In Example 1, Cl can preliminary prevent the accident of the car and the train, but the car can reenter the railroad, for instance, in the action trace $\langle \mathrm{Ca}, \mathrm{Cc}, \mathrm{Cl}, \mathrm{Ca}, \mathrm{Cc}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc} \rangle$. Cl occurs after Cc in the action trace but the property violation is not prevented. Since a loop in the underlying execution of an action trace violates minimality, we can remove the loop in the execution and obtain an action trace $\langle \mathrm{Ca}, \mathrm{Cc}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc} \rangle$ that violates the property. In the following, we define a cause for simple bad traces. We can give a cause for a non-simple bad trace by removing loops in the underlying execution. The resulting execution has a simple bad trace for which we define a cause.

In order to describe a cause with non-occurrence events, we need to express also the events that are not allowed to occur, and their order with the other events in the cause. We cannot describe, for instance, that Cl does not occur after Cc by a strict partial order. Instead, we describe causes with non-occurrence in a subset of the EOL logic in Definition 26 that we call *simple event order logic* (sEOL) logic. An sEOL formula ψ has an event set $\mathcal{A}$, and in the formula an event a either has to occur (a), or is not allowed to occur ($\neg a$). ψ describes the orderings between event pairs by the $\barwedge$-operator, and conjoins several constraints using the $\wedge$-operator. Remember that $\barwedge$ binds stricter than $\wedge$. The semantics of sEOL is the semantics of EOL given in Definition 9.

Definition 26 (Syntax of Simple Event Order Logic (sEOL))**.** *An sEOL formula ψ over a set $\mathcal{A}$ of event variables is formed according to the following grammar:*

$$\psi ::= \neg a \mid a \mid \psi \barwedge \psi \mid \psi \wedge \psi$$

where $a \in \mathcal{A}$.

We can express the cause ϕ_r of Example 2 by the sEOL formula

$$\psi_r^{\phi} = \mathrm{Ca} \barwedge \mathrm{Cc} \wedge \mathrm{Ta} \barwedge \mathrm{Tc} \wedge \mathrm{Gf} \barwedge \mathrm{Tc}. \tag{4.1}$$

This formula allows different orders of Cc and Tc. Whenever Cc occurs before Tc, Cl can prevent the violation of φ from occurring. In case Tc occurs before Cc, Tl can prevent the violation of φ. We extend the sEOL formula ψ_r^{ϕ} to prevent these two non-occurrence events, and obtain the sEOL formulae

$$\psi_r^1 = \mathrm{Ca} \barwedge \mathrm{Cc} \wedge \mathrm{Ta} \barwedge \mathrm{Tc} \wedge \mathrm{Gf} \barwedge \mathrm{Tc} \wedge \mathrm{Cc} \barwedge \neg\mathrm{Cl} \barwedge \mathrm{Tc} \tag{4.2}$$

and

$$\psi_r^2 = \mathrm{Ca} \barwedge \mathrm{Cc} \wedge \mathrm{Ta} \barwedge \mathrm{Tc} \wedge \mathrm{Gf} \barwedge \mathrm{Tc} \wedge \mathrm{Tc} \barwedge \neg\mathrm{Tl} \barwedge \mathrm{Cc}. \tag{4.3}$$

For a formal definition of a cause, we need an sEOL formula comparison to ensure that the sEOL formulae ψ_r^1 and ψ_r^2 are minimal. We compare two sEOL formulae in the number of satisfying action traces by the *subsumption relation* in Definition 27. For instance, every action trace where a occurs before b satisfies $a \barwedge b$, and also satisfies $a \wedge b$. Additionally, the action trace $\langle b, a\rangle$ satisfies $a \wedge b$ but does not satisfy $a \barwedge b$, hence, $a \wedge b \subset a \barwedge b$ holds. The more action traces satisfy an sEOL formula the fewer restrictions are described by an sEOL formula.

Definition 27 (Subsumption Relation)**.** *An sEOL formula ψ with an event set $\mathcal{A}$ is a subsumption of another sEOL formula ψ' with an event set $\mathcal{A}'$, denoted by $\psi \subseteq \psi'$, iff for every action trace σ of any transition system, if $\sigma \vDash_e \psi'$ then $\sigma \vDash_e \psi$. ψ is a true subsumption of ψ', denoted by $\psi \subset \psi'$, iff $\psi \subseteq \psi'$ and there exists a transition system with an action trace σ' such that $\sigma' \vDash_e \psi$ and $\sigma' \nvDash_e \psi'$ holds.*

We give some examples for subsumption relations between sEOL formulae with the events a, b and c. The sEOL formula a is a true subsumption of the sEOL formulae $a \barwedge c$, denoted by $a \subset a \barwedge c$, because every action trace σ satisfying $a \barwedge c$ contains also the action a. Another example is that the sEOL formula $a \barwedge \neg b$ is a true subsumption of the sEOL formula $a \wedge \neg b$, since the action trace $\langle b, a\rangle$ satisfies $a \barwedge \neg b$ but does not satisfy $a \wedge \neg b$. The sEOL formula $b \wedge a \barwedge c$ is not a subsumption of the sEOL formula $a \wedge b \barwedge c$ since the formulae are satisfied with different action traces. Both formulae are satisfied by the action trace $\langle a, b, c\rangle$ but the action trace $\langle a, c, b\rangle$ satisfies $b \wedge a \barwedge c$ but not $a \wedge b \barwedge c$, and the action trace $\langle b, c, a\rangle$ satisfies $a \wedge b \barwedge c$ but not $b \wedge a \barwedge c$. Thus, two sEOL formulae that are not subsumptions of another can both be satisfied by the same action trace.

We introduce sEOL to describe a cause with non-occurrence and are now prepared to give a cause definition in Definition 28 that ensures regularity for transition systems. Regularity is ensured by condition CC2(2). This condition is satisfied iff for every action trace that satisfies the cause ψ, the property violation occurs. CC2(2) in Definition 28 is more strict than CC1.1 of Definition 22, because when every action trace σ'' with $\sigma'' \vDash_e \psi$ violates φ, then also every ordered sequence σ violates φ. Thus, we relax CC1.1 in Definition 28, since whenever σ is an action trace of T satisfying ψ, CC2(2) ensures that it is also a bad trace. CC1 refers now to a subset of $\mathcal{A}$ since several events in $\mathcal{A}$ can be non-occurrence events that are not allowed to occur. CC2(1) in Definition 28 is equivalent to CC2 in Definition 22 and ensures the counterfactual argument. CC3 in Definition 28 utilizes the suborder relation to ensure minimality.

Definition 28 (Cause for a Regular Safety Property Violation)**.** *Let* $T = (S, Act, \rightarrow, I, AP, L)$ *be a transition system that deterministically violates a regular safety property* φ*. An sEOL formula* ψ *with an event set* $\mathcal{A} \subseteq Act$ *is a* cause *for the violation of* φ*, iff the following conditions are satisfied:*

CC1 *Any ordered sequence* σ *of a subset of* $\mathcal{A}$ *with* $\sigma \vDash_e \psi$

1. *is an action trace of* T*, and*
2. *no true subtrace* $\sigma' \sqsubset \sigma$ *violates* φ*.*

CC2(1) *There exists an action trace* σ'*, such that* $\sigma' \nvDash_e \psi$ *and* $\sigma' \vDash \varphi$*.*

CC2(2) *Every action trace* σ'' *of* T *for which* $\sigma'' \vDash_e \psi$ *holds, violates* φ*.*

CC3 ψ *is minimal, i.e., no true subsumption of* ψ *satisfies CC1, CC2(1) and CC2(2).*

We now show that ψ_r^1 in Formula 4.2 with an event set $\mathcal{A}_{\psi_r^1}$ is a cause according to Definition 28 for Example 1.

- CC1 is fulfilled since the linear orders of $\mathcal{A}_{\psi_r^1}$ that satisfy ψ_r^1 are identical to the action traces given in Example 2 that satisfy $\mathrm{Cc} \wedge \mathrm{Tc}$:

  Ca Cc Ta Gf Tc, Ta Ca Gf Cc Tc, Ca Gf Cc Ta Tc, Ca Ta Cc Gf Tc,
  Gf Ca Ta Cc Tc, Gf Ca Cc Ta Tc, Ta Ca Cc Gf Tc, Ta Gf Ca Cc Tc,
  Ca Ta Gf Cc Tc, Gf Ta Ca Cc Tc, Ca Gf Ta Cc Tc, Ca Cc Gf Ta Tc,

  Notice, that all action traces in this set are minimal, i.e., they do not contain subtraces that violate φ.

- CC2(1) is fulfilled since the action trace

  $$\sigma' = \langle \mathrm{Ca},\ \mathrm{Ta},\ \mathrm{Gf},\ \mathrm{Tc} \rangle$$

  is an action trace of T with $\sigma' \nvDash \psi_r^1$ and $\sigma' \vDash \varphi$.

- ψ_r^1 satisfies CC2(2), if no action trace exists that satisfies ψ_r^1 and φ. In an action trace that satisfies ψ_r^1, the car enters the crossing and the train enters the crossing. The property violation occurs when the car and the train are concurrently in the crossing. In Example 1, the car can only leave the crossing by action Cl. In every action trace that satisfies ψ_r^1, a car that has entered is not allowed to leave (Cl) the crossing. Consequently, ψ_r^1 satisfies CC2(2).

- ψ_r^1 is minimal if no formula is a cause and a subsumption of ψ_r^1. Notice, that a subsumption formula is satisfied by more action traces and, hence, either has fewer events or fewer order constraints.

  ψ_r^1 contains the events of ϕ_r that are necessary for the property violation to occur, and the event Cl that cannot be removed without violating CC2(2). Hence, every event in ψ_r^1 is causal.

  The order constraints in ϕ_r are causal, as we have previously shown. The only other order constraints in ψ_r^1 are added by $\mathrm{Cc} \wedge \neg\mathrm{Cl} \wedge \mathrm{Tc}$. A substitution

of $\mathrm{Cc} \barwedge \neg\mathrm{Cl} \barwedge \mathrm{Tc}$ in ψ_r^1 with $\neg\mathrm{Cl} \barwedge \mathrm{Tc}$ does not result in a formula that is a cause since the substitution leads to a formula that is satisfied by the action trace $\langle \mathrm{Ca}, \mathrm{Cl}, \mathrm{Cc}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc} \rangle$, which is not a valid action trace and, hence, violates CC1. A substitution of $\mathrm{Cc} \barwedge \neg\mathrm{Cl} \barwedge \mathrm{Tc}$ in ψ_r^1 with $\mathrm{Cc} \barwedge \neg\mathrm{Cl}$ is not a formula that subsumes ψ_r^1 since the result is a formula that is not satisfied by the action trace $\langle \mathrm{Ca}, \mathrm{Cc}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Cl} \rangle$ but which satisfies ψ_r^1. Thus, ψ_r^1 is minimal and CC3 is satisfied.

Because ψ_r^1 satisfies CC1, CC2(1), CC2(2) and CC3, ψ_r^1 is a cause according to Definition 28.

Ensuring CC2(2) by Analyzing Non-Occurrence. A cause ϕ according to Definition 22 can be rewritten as an sEOL formula ψ. By construction, ψ satisfies CC1 and CC2(1) of Definition 28 because the corresponding conditions in Definition 22 are stricter. We now present an approach to ensure CC2(2) for ψ. ψ either satisfies CC2(2) and is a cause according to Definition 28, or otherwise an action trace $\hat{\sigma}$ exists where $\hat{\sigma} \vDash_e \psi$ and $\hat{\sigma} \vDash \varphi$ holds. Let $\mathcal{A}_\psi$ be the event set of ψ. In case $\hat{\sigma} = \langle a_1 \dots a_n \rangle$ exists, CC2(2) does not hold and we adapt Definition 13 in Definition 29 to compute the set $\mathcal{A}_{non}$ of non-occurrence events that are in $\hat{\sigma}$ but not in $\mathcal{A}_\psi$. Every event in $\mathcal{A}_{non}$ is necessary to prevent the satisfaction of ψ since by definition no subtrace of $\hat{\sigma}$ is a good trace that satisfies ψ. For every action $a_i \in \mathcal{A}_{non}$, we create a constraint $C_i = a_k \barwedge \neg a_i \barwedge a_l$ where the actions $a_k, a_l \in \mathcal{A}_\psi$, $a_k, a_l \in \hat{\sigma}$, k is the maximal index in $\hat{\sigma}$ where $k < i$, and l the minimal index where $i < l$. In case a_k does not exist, we construct a constraint $\neg a_i \barwedge a_l$, and in case a_l does not exists, we construct a constraint $a_k \barwedge \neg a_i$. Remember, that every event in $\mathcal{A}_{non}$ is necessary to prevent the property violation. Hence, we construct a cause $\psi \wedge C_i$ for every a_i.

Definition 29 (Non-Occurrence Events)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system and let ψ be a cause with the event set $\mathcal{A}_\psi$ such that ψ satisfies the conditions CC1, CC2(1) and CC3 but not CC2(2) of Definition 28 for the violation of a regular safety property φ. Assume an action trace $\hat{\sigma}$ of T with $\hat{\sigma} \vDash_e \psi$ and $\hat{\sigma} \vDash \varphi$ such that for no subtrace $\hat{\sigma}' \sqsubset \hat{\sigma}$ holds $\hat{\sigma}' \vDash_e \psi$ and $\hat{\sigma}' \vDash \varphi$. The events in $\mathcal{A}_\sigma$ that are not in $\mathcal{A}_\psi$ are* non-occurrence events.

We now apply this procedure to the railroad example in Example 1 and the sEOL formula ψ_r^ϕ in Formula 4.1. ψ_r^ϕ is not a cause according to Definition 28 because the action trace $\langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Cc}, \mathrm{Cl}, \mathrm{Tc} \rangle$ satisfies ψ_r^ϕ and the property. Hence, we conjunct the constraint $\mathrm{Cc} \barwedge \neg\mathrm{Cl} \barwedge \mathrm{Tc}$ with ψ_r^ϕ and the result is ψ_r^1 in Formula 4.2.

$\mathrm{Cc} \barwedge \neg\mathrm{Cl} \barwedge \mathrm{Tc}$ in ψ_r^1 implies $\mathrm{Cc} \barwedge \mathrm{Tc}$ and prevents that the Tc occurs before Cc which is possible for an action trace satisfying ψ_r^ϕ. For instance, the action trace $\langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Cc} \rangle$ satisfies ψ_r^ϕ but not ψ_r^1. As we see by this example, when we change a cause ψ into ψ^1 to prevent non-occurrence events, a causal trace σ can exist that satisfies ψ but not ψ^1. Note, that by Definition 29 the set of non-occurrence events is disjoint with the set of events in ψ. Hence, $\sigma \nvDash_e \psi^1$ because of the order relation $a_k \barwedge a_l$, and we construct $\psi^2 = \psi \wedge a_l \barwedge a_k$ as another cause. For ψ^2, $\sigma \vDash_e \psi^2$ holds by construction.

For Example 1, we construct a second cause $\psi_r^{2\phi} = \psi_\phi^r \wedge \mathrm{Tc} \barwedge \mathrm{Cc}$. $\psi_r^{2\phi}$ violates CC2(2) of Definition 28 since the action trace $\langle \mathrm{Ca}, \mathrm{Ta}, \mathrm{Gf}, \mathrm{Tc}, \mathrm{Tl}, \mathrm{Cc} \rangle$ satisfies $\psi_r^{2\phi}$ but satisfies the property. Hence, we conjoin $\psi_r^{2\phi}$ with $\mathrm{Tc} \barwedge \neg \mathrm{Tl} \barwedge \mathrm{Cc}$ and result in ψ_r^2 given in Formula 4.3.

Causality Checking based on Definition 28 computes a cause ψ according to Definition 28 without satisfying AC2(2) for each bad trace with a simple execution of a transition system and combines these causes into a cause set Ψ. For a cause ψ in Ψ, Causality Checking searches for every good trace $\hat{\sigma}$ with a simple execution where $\hat{\sigma} \vDash_e \psi$ holds. Since $\hat{\sigma} \vDash_e \psi$ contradicts that ψ is a cause, Causality Checking determines the non-occurrence events (c.f. Definition 29) in $\hat{\sigma}$, creates a cause ψ^1 and ψ^2 as described above and replaces ψ in Ψ with ψ^1 and ψ^2. In case $\hat{\sigma}$ has an event set $\mathcal{A}_{\hat{\sigma}}$ that is equivalent to the event set $\mathcal{A}_\psi$ of ψ, Causality Checking found that the underlying transition system T does not deterministically violate the given property φ and Causality Checking based on Definition 22 is applied to T. Otherwise, T deterministically violates φ. After every bad trace is determined, Causality Checking removes every cause in Ψ that is not minimal since another cause in Ψ is a subsumption of it by Definition 27. Finally, Causality Checking returns Ψ.

Completeness and Soundness of Causality Checking Based on Definition 28. We now prove Theorem 4 that Causality Checking based on Definition 28 is complete and Theorem 5 that Causality Checking based on Definition 28 is sound. In order to prove Theorem 4, we need Lemma 3 that a cause according to Definition 28 exists for every simple bad trace of a TS that deterministically violates or satisfies a given property. For a not simple bad trace, we create a simple bad trace by removing loops in the underlying execution as mentioned before.

Lemma 3 (Existence of Cause with Non-Occurrence)**.** *Let $T = (S, Act, \rightarrow, I, AP, L)$ be a transition system that deterministically violates φ and where any state in I satisfies the regular safety property φ. For each simple action trace σ of T with $\sigma \nvDash \varphi$, there exists a cause ψ by Definition 28 with $\sigma \vDash_e \psi$.*

Proof. Let $\sigma := \langle a_0 \dots a_n \rangle$ be an simple action trace in a TS T that deterministically violates a regular safety property φ, and no initial state of T violates φ. For σ, we can construct an sEOL formula $\psi := (a_0 \barwedge a_1) \wedge \dots \wedge (a_{n-1} \barwedge a_n)$ with $\mathcal{A} := \{a_0, \dots, a_n\}$. Notice, ψ is only a different notation of the strict partial order representing a cause in Lemma 1, hence, CC1 and CC2(1) hold by the proof of Lemma 1. For ψ, CC2(2) is not satisfied iff an action trace $\hat{\sigma}$ exist with $\hat{\sigma} \vDash_e \psi$ and $\hat{\sigma} \vDash \varphi$. At least one non-occurrence event a_i according to Definition 29 has to exist for $\hat{\sigma}$ since T deterministically violates φ. We conjoin ψ with a constraint of the form $a_k \barwedge \neg a_i \barwedge a_l$ with $a_k, a_l \in \mathcal{A}_\psi$. Because of this conjunction, $\hat{\sigma} \nvDash_e \psi$ holds.

Since T contains only finitely many actions, there exist only finitely many non-occurrence events that can be prevented between finitely many events in ψ. Hence, after finitely many iterations no $\hat{\sigma}$ exists with $\hat{\sigma} \vDash_e \psi$ and $\hat{\sigma} \nvDash \varphi$ and, hence, CC2(2) is satisfied.

For σ, a cause exists because either ψ is minimal (CC3) or a cause ψ' exists that satisfies $\psi' \subset \psi$ and the conditions CC1 to CC3 of Definition 28. In both cases, σ satisfies a cause. □

We prove completeness of Causality Checking based on Definition 28 in Theorem 4 for the simple bad traces of a transition system.

Theorem 4 (Causality Checking Based on Definition 28 is Complete)**.** *Assume a transition system T that deterministically violates a regular safety property. Causality Checking based on Definition 28 is complete if it returns a cause set that is complete by Definition 24 for the simple bad traces of T.*

Proof. Causality Checking considers every bad trace σ of a simple counterexample of a transition system T and adds a cause ψ where $\sigma \vDash_e \psi$ holds and that exists by Lemma 3 to Ψ. For every simple bad trace of T, is by construction a cause in Ψ. □

Theorem 5 (Causality Checking According to Definition 28 is Sound)**.** *Assume a transition system T that deterministically violates a regular safety property φ. Causality Checking based on Definition 28 is sound if it returns for T and φ, a sound cause set (see Definition 25).*

Proof. Let Causality Checking return a cause set Ψ for a transition system T that deterministically violates a regular safety property φ. Assume a cause ψ in Ψ for which no bad trace σ in T with $\sigma \vDash_e \psi$ exists, or that is not a cause according to Definition 28.

Causality Checking adds ψ only to Ψ, when ψ was computed for a bad trace σ of T such that $\sigma \vDash_e \psi$. Hence, σ with $\sigma \vDash_e \psi$ exists.

Since σ exists, the other case has to hold that ψ is not a cause since it does not satisfy one of the conditions CC1, CC2(1), CC2(2) or CC3 of Definition 28. CC1 and CC2(1) were fulfilled at the time ψ was added to Ψ, and these conditions cannot falsify by adding another cause to Ψ. CC2(2) is not satisfied when an action trace $\hat{\sigma}$ exists in T with $\hat{\sigma} \vDash_e \psi$ and $\hat{\sigma} \vDash \varphi$. In case $\hat{\sigma}$ would exist, Causality Checking determines the non-occurrence events according to Definition 29 and disregards one of the non-occurrence events such that $\hat{\sigma} \nvDash_e \psi$. Since the number of potential non-occurrence events is finite, ψ will ultimately satisfies CC2(2). CC3 is satisfied since if a cause ψ' was a subsumption of ψ, then Causality Checking would remove ψ from Ψ.

Hence, the assumed ψ cannot exist in Ψ, and Causality Checking is sound. □

We proved that Causality Checking is complete and sound based on Definition 22 and Definition 28. A significant change from Definition 12 is that the action traces are compared by a subtrace relation (see CC1) instead of a subset relation. We present algorithms in Chapter 5 and Chapter 6 that implement Causality Checking based on the new cause definitions.

Chapter 5

An Algorithm for Computing Causal Trace Sets

Contents

The content of this part is based on the publication [87].

5.1 Introduction

Causality Checking analyzes the action traces of a system to find the causal traces that are necessary to compute the causes of a system. It has been implemented in the SpinCause tool [106], which computes causes for SPIN [69] models. In this chapter, we motivate and describe a new implementation of Causality Checking and we update SpinCause to this implementation.

In this chapter, we will see that the originally implemented algorithm under-approximates the causal traces in a system and, hence, cannot find every cause for a hazard in a system. We also present complete algorithms to compute every causal trace of a cause according to Definition 22 and Definition 28. A significant difference from the original algorithm is the use of a subtrace relation to compare causal traces instead of a subset relation over actions.

An overview of the new Causality Checking approach is described by the BPMN workflow [82] in Figure 5.1. For a given TS and property, an algorithm computes all

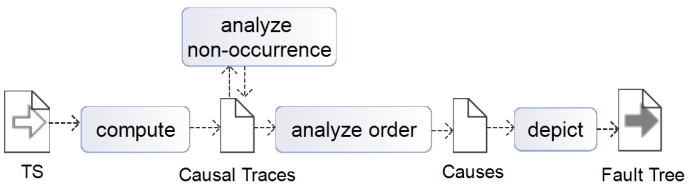


Figure 5.1: BPMN Workflow of the Causality Checking Implementation

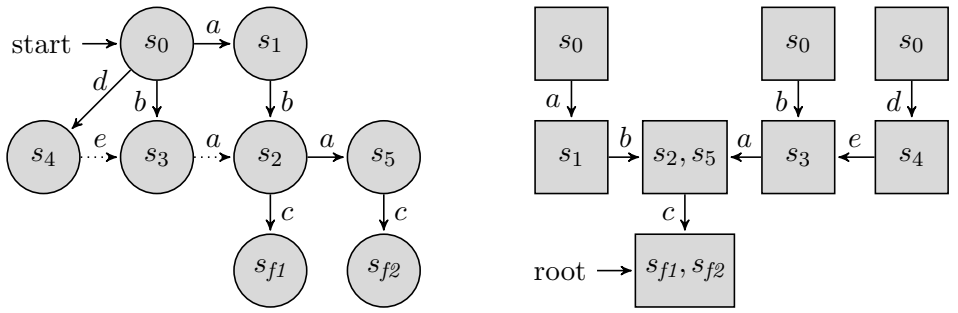
the causal traces according to Definition 22. If we additionally apply an algorithm to analyze non-occurrence, Causality Checking computes the causal traces according to Definition 28. The causal traces are the input to the algorithms in Chapter 6 that compute causes and depict these causes in a fault tree. For causal traces without non-occurrence events, Causality Checking computes causes according to Definition 22, otherwise, it computes causes according to Definition 28.

Causality Checking utilizes a reachability analysis to explore the state space defined by a considered model and to find violations of a regular safety property. A state space exploration is typically implemented using a depth-first (DFS) or breadth-first search (BFS). When the state space exploration reaches a state that violates the property, a counterexample leading into this state will be generated. In the implementations of Causality Checking, the state space traversal is implemented using a modified model checking algorithm. The modifications concern the fact that in Causality Checking, we need to explore every simple bad trace of a model in order to find every cause for a property violation.

Pivotal for the performance of Causality Checking is the computation and storage of all bad traces found during state space exploration. The existing implementation of Causality Checking [106] use a prefix tree data structure to store system executions. A crucial aspect in this regard is how the state space exploration deals with the situation in which a state s is visited that has been visited before during the search. s is then referred to as a duplicate state. Both a BFS and a DFS would not further explore a duplicate state, since doing so would lead to an exponential time penalty. However, when performing Causality Checking all executions need to be explored, irrespective of whether they contain a state which, due to the search strategy employed, happens to be a duplicate state. As a consequence, during the exploration of the system executions, it is necessary to concatenate the prefixes leading from the initial system state into a duplicate state with all possible suffixes starting with the duplicate state. This should be done efficiently, in particular by behaving benevolently on practical examples in the face of a potential exponential time and space penalty.

For performance reasons, the algorithm *Duplicate State Prefix Matching (DSPM)* that is implemented in SpinCause and QuantUM [102] under-approximates the computation of the execution suffixes beginning in duplicate states. It does not guarantee that all of them will be considered. The objective of this chapter is to propose an algorithm that deals with duplicates precisely, without relying on an approximation, and which is nonetheless efficient. The algorithm that we propose, which we refer to as *Causal Trace Backward Search (CTBS)*, performs an efficient exploration and analysis of all counterexamples found during the state space exploration. It works on-the-fly and returns preliminary results at any point during the computation. While DSPM generates valuable approximations for realistic models, as we shall see, it is incomplete for Causality Checking, whereas CTBS is complete.

A Motivating Example. Consider the transition system depicted in Figure 5.2(a). Causality Checking conceptually works on transition systems, which can be automatically derived from higher-level modeling languages, such as SysML. Assume that states s_{f1} and s_{f2} violate a uniquely defined reachability property φ, corre-



(a) Transition System (b) BCT tree with 3 minimal bad traces

Figure 5.2: Transition System Analysis with Duplicate States

sponding to the occurrence of a property violation in the underlying domain model. We henceforth refer to these states as failure states. In the context of Causality Checking, reaching a failure state corresponds to the effect for which we compute a cause.

The action traces $\langle a, b, c\rangle$, $\langle b, a, c\rangle$ and $\langle d, e, a, c\rangle$ are minimal bad traces for the violation of φ. They are minimal since they do not contain a non-contiguous subtrace that also is a bad trace of φ. In this sense, $\langle a, b, a, c\rangle$ is a non-minimal bad trace, since it contains $\langle a, b, c\rangle$ as a non-contiguous subtrace, formally $\langle a, b, c\rangle \sqsubset \langle a, b, a, c\rangle$. The set of all minimal bad traces is a causal trace set formalized in Definition 30.

A standard DFS or BFS, modified to compute every action trace, would not return the action trace $\langle b, a, c\rangle$ in case it had previously explored the action trace $\langle a, b, c\rangle$, since state s_2 would then be a duplicate state. We therefore need to ensure that a state space exploration concatenates all action trace prefixes starting in the initial state and leading into any duplicate state with all action trace suffixes that start in the respective duplicate state. In the example, this means that the prefix $\langle b, a\rangle$, for instance, needs to be concatenated with the suffixes $\langle c\rangle$ and $\langle a, c\rangle$. Notice that the resulting action trace $\langle b, a, a, c\rangle$ will not be included in the causal trace set since it is not a minimal bad trace.

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

duplicate states.

Structure of the Chapter. We formalize in Section 5.2 the causal trace set as a set of the causal traces proposed in Definition 22. In Section 5.3, we present the algorithm proposed in [102] to compute a causal trace set, propose a new algorithm and compare the computational complexity of both algorithms. We qualitatively and quantitatively compare the algorithms by several case studies in Section 5.4. In Section 5.5, we draw conclusions.

5.2 Formalization of a Causal Trace Set

A causal trace by Definition 22 is a minimal action sequences satisfying a cause. We combine the causal traces that are members of different causes that explain the violation of the same property in a given system model to a *causal trace set* and formalize this set in Definition 30.

A causal trace set relies on two essential properties of the causal traces included in the set. First, every bad trace in a system model needs to be represented by a causal trace of the causal trace set (completeness), and second, the causal trace set contains only minimal bad traces.

Definition 30 (Causal Trace Set)**.** *Assume a TS T and a safety property φ. Let σ and σ' be action traces in T. A causal trace set is a subset Υ of the action traces of T that satisfies the following conditions:*

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

We now state in Theorem 6 that a causal trace set contains exactly the causal traces of causes by Definition 22 for a given TS and a given regular safety property.

Theorem 6. *Assume a causal trace set Υ and a set Φ of causes according to Definition 22, both for a given TS and a given regular safety property. An action trace σ is in Υ iff σ is also a causal trace of a cause in Υ.*

Proof. Assume a causal trace set Υ for a TS T and a regular safety property φ, and a cause set Φ with the causes satisfying Definition 22 for T and φ. An action trace σ is a causal trace of a cause ϕ with the event set $\mathcal{A}$ iff σ is an ordered sequence of $\mathcal{A}$ consistent with ϕ. We split the proof of the theorem in two cases. In a first case $\Rightarrow$, we assume an action trace σ in Υ and show that σ is also a causal trace of a cause ϕ in Φ, and in a second case $\Leftarrow$, we assume a causal trace σ of a cause ϕ in Φ and show that $\sigma \in \Upsilon$ holds.

- $\Rightarrow$ We assume an action trace σ with an action set $\mathcal{A}_\sigma$ in Υ, and we want to show that σ is a causal trace of a cause ϕ in Φ. For σ, $\sigma \not\models \varphi$ holds by TC1 and no subtrace $\sigma' \sqsubset \sigma$ exists with $\sigma' \in \Upsilon$ because of TC2. Thus, no subtrace

of σ violates φ. Due to Lemma 1, a cause ϕ with an event set $\mathcal{A}$ exists such that $\sigma \vDash \phi$ holds. Since no subtrace of σ violates φ and $\sigma \vDash \phi$, $\mathcal{A} = \mathcal{A}_\sigma$ holds. Because σ is an ordered sequence of $\mathcal{A}$ and $\sigma \vDash \phi$ holds, we conclude that σ is a causal trace of ϕ.

$\Leftarrow$ We assume a cause ϕ with an event set $\mathcal{A}$ in Φ and a causal trace σ of ϕ. A causal trace is an action trace of T that violates φ and no true subtrace $\sigma' \sqsubset \sigma$ violates φ (CC1). According to TC1, an action trace σ' exists for σ in Υ that is a subtrace $\sigma' \sqsubseteq \sigma$ and violates φ. Since by CC1 σ' cannot be a true subtrace, $\sigma' = \sigma$ has to hold. Thus, σ is in Υ.

Since both cases hold, we conclude that an action trace is in Υ iff it is a causal trace of a cause in Φ. □

5.3 Algorithms for Computing a Causal Trace Set

We now present the DSPM algorithm originally used in Causality Checking to compute a causal trace set. We then present the complete CTBS algorithm. Both algorithms utilize state space exploration algorithms that are modified for the purpose of Causality Checking in two fundamental ways. First, the state space exploration continues after a first property violating state is found. Second, when reaching a duplicate state, an execution called a *duplicate execution* reaching this duplicate state is returned by the state space exploration.

5.3.1 Duplicate State Prefix Matching Algorithm (DSPM) [102]

We give the idea of the DSPM algorithm [102] before we explain the algorithm in more detail. When the state space exploration encounters a counterexample or a duplicate execution, it hands the execution over to the DSPM algorithm. The DSPM algorithm compares all counterexamples with each other and stores only minimal counterexamples. Duplicate executions are concatenated with previously stored minimal counterexamples which contain the respective duplicate state. For the modified BFS algorithm, duplicate executions are cached in a list until the BFS algorithm terminates, and then concatenated.

Notice that the DSPM algorithm in [102] uses a subset relation over actions to compare action traces because it is based on Definition 12. We adapt the DSPM algorithm for Causality Checking based on Definition 22 by replacing the subset comparison in the algorithm by a subtrace comparison as motivated in Chapter 4.2.

Pseudo Code of Adapted DSPM [102]. We describe in detail the implementation of the DSPM algorithm in [102] given by the pseudo code in Listing 5.1. The algorithm uses four data structures: A list `redTraces` stores the execution of causal traces, the prefix trees `prefix_actions` and `prefix_states` store executions, and a map `matchList` enables an efficient lookup of executions that contain a certain state.

```
1   List<Execution> redTraces;
2   PrefixTree prefix_actions;
```

```
3     PrefixTree prefix_states;
4     HashMap<State, List<Execution>> matchList;
5
6     function addTrace(Execution e)
7        // add execution to the prefix trees
8        addTo(prefix_actions, e);
9        addTo(prefix_states, e);
10       // insert all states into the match
11       addAllStatesTo(matchList, e);
12       if !e.isBad()
13          return;
14       FOR EACH Execution e' in redTraces
15          if e is subtrace of e'
16             //e describes a subtrace of a causal trace
17             redTraces.remove(e');
18          else e' is subtrace of e
19             //causal trace in e' is subtrace of e
20             return;
21       addTo(redTraces, e);
22
23    function matchDuplicateState(State s, Execution e)
24      FOR EACH Execution e' in matchList.getTraceFor(s)
25       //append suffix of e' to e
26       Execution e'' = replacePrefix(s, e, e');
27       addTrace(e'');
```

Listing 5.1: Sketch of Adapted DSPM Algorithm [102]

The function `addTrace` is called for every execution e that is found during the state space exploration. The function first enters the states and actions defined by e into the prefix trees (line 7-9), and adds all states of e to the `matchList` (line 11). In case, e is not a counterexample the function exits. Otherwise, e is a counterexample and the action trace t in e is compared with the action trace t' in every execution e' in `redTraces` (line 14). If t is a subtrace of t' (line 15), e' is no longer minimal and removed from `redTraces` (line 17). In case the t' is a subtrace of t, then e is not minimal and the function exits (line 18-20). Finally, when t is not a subtrace of any causal trace and then t is a causal trace itself and e is added to `redTraces` (line 21).

The function `matchDuplicateState` is called for every duplicate execution $s_0 a_x \dots s$ with a duplicate state s. It obtains all executions that contain the duplicate state s, using a lookup on the map `matchList` (line 24). For every execution $s_0 a_0 \dots s \dots s_n$, the function `replacePrefix` concatenates the execution suffix $s \dots s_n$ with the duplicate execution $s_0 a_x \dots s$ (line 26). The function `addTrace` is called with the resulting execution $s_0 a_x \dots s \dots s_n$ (line 27) to check whether it is causal. Unless all duplicate executions are processed, the set of causal traces described by `redTraces` can miss some causal traces and contain longer not causal action traces.

As discussed above, DSPM computes a potentially incomplete causal trace set when it is possible to reach a duplicate state via another duplicate state. The order of processing duplicates depends on the search order used by the state space explo-

ration algorithm. As explained above, the order of encountering duplicate states and their processing may lead to an incompleteness in the discovery of causal traces. It is not obvious whether there is an ordering that would avoid this incompleteness. In particular, ordering the processing of duplicate states according to the length of the action trace needed to reach them will not solve the problem, as we found out.

5.3.2 Causal Trace Backward Search Algorithm (CTBS)

The CTBS algorithm computes minimal bad traces using a *Backwards Causal Trace* (BCT) tree data structure. Consider the example BCT tree depicted in Figure 5.2(b) which is derived from the transition system in Figure 5.2(a). CTBS is interleaved with the state space exploration algorithm. When the state space exploration encounters an execution corresponding to a counterexample, to reaching a non-property violating action trace ending in a terminal state or to reaching a duplicate state, the corresponding execution will be handed over to CTBS. CTBS maintains the BCT tree data structure, which is implemented as a prefix tree. The edges of the BCT are labeled with actions. The root vertex of this tree is labeled with all failure states of the system, in the example with the states s_{f1} and s_{f2}. The non-root vertices are labeled with a set of states. These states are equivalent in the sense that one can reach one of the failure states from them via an identical action trace. The action trace is defined by the sequence of action labels along the edges of the BCT tree that are encountered on the path from the considered vertex to the root vertex. As an example, the vertex labeled s_2, s_5 implies that from states s_2 and s_5 a failure state can be reached via an action trace $\langle c \rangle$. Notice that such action traces correspond to suffixes of counterexamples. When a suffix contains an initial state, then it represents a causal trace. As an example consider the action trace $\langle a, b, c \rangle$ leading from the leaf vertex s_0 to the root vertex.

The CTBS algorithm is designed to satisfy two major requirements:

1) It needs to ensure that all action traces in the system are completely analyzed, independently of the search order during the state space exploration.

2) The algorithm should be efficient wrt. both space and time, in particular by storing only minimal bad traces and by ignoring non-minimal bad traces.

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

In order to ensure efficiency, the algorithm exploits the observation that for each state s in a minimal counterexample, there is no shorter action trace to reach

a failure state from s than the suffix of the action trace corresponding to the counterexample that starts in s. This implies that all non-minimal suffixes can be removed from the BCT tree, as expressed by the prune rules *PR1* and *PR2*. To illustrate this point, assume that a given BCT tree contains a path b, a, c and that the state space exploration hands the action trace $\langle a, b, a, c\rangle$ over to CTBS. Assume further that $\langle a, b, a, c\rangle$ will be split into transition triples and integrated into the BCT tree. Since b, a, c is contained in $\langle a, b, a, c\rangle$, the vertex that stores $\langle a, b, a, c\rangle$ and the complete subtree that hangs off it will be pruned from the BCT tree.

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

Lemma 4 shows that the path of a vertex deleted by prune rule PR1 is not a suffix of a minimal bad trace.

Lemma 4. *A suffix that contains a shorter bad trace is not a suffix of a minimal bad trace.*

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

A second prune rule removes a state label from a vertex when another vertex labeled with this state has a shorter action trace to reach a failure state. Assume a vertex labeled with s_2 to be in the BCT tree of Figure 5.2(b) from which the root vertex can be reached via action trace $\langle a, c\rangle$. The BCT tree contains a vertex labeled $s_2 s_5$ which has an action trace $\langle c\rangle$. The action trace of the assumed vertex labeled s_2 contains the action trace of vertex labeled $s_2 s_5$. As a consequence, prune rule PR2 removes the state label s_2 from the respective vertex and thereby removes the longer suffix $\langle a, c\rangle$ from the BCT tree.

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

Lemma 5 shows that a bad trace with a suffix that is equivalent to the action trace of the vertex v in *PR2* cannot be minimal.

Lemma 5. *Any bad trace with a suffix starting in a state s in vertex v is not minimal when another vertex v' with state s exists and the action trace t_v contains action trace $t_{v'}$.*

Proof. Assume two different vertices v and v' with state s and with action traces t_v and $t_{v'}$ that satisfy $t_{v'} \sqsubseteq t_v$, and an arbitrary bad trace t_c with suffix t_v. The bad trace t_c is not minimal since we can construct a shorter bad trace than t_c. Notice that $t_v \neq t_{v'}$ otherwise $v = v'$. We split t_c in state s in a prefix $s_0 a_0 \ldots s$ and the suffix $t_v = s \ldots s_f$, prepend the prefix to the suffix $t_{v'} = s \ldots s_{f'}$ and result with a

counterexample $s_0 a_0 \dots s \dots s_{f'}$ with a bad trace t'_c. The assumption $t_{v'} \sqsubseteq t_v$ with $t_v \neq t_{v'}$ results in $t_{v'} \sqsubset t_v$. $t'_c \sqsubset t_c$ holds because the prefixes of t_c and t'_c before state s are equivalent by construction and for the suffixes $t_{v'} \sqsubset t_v$ holds. Thus, t'_c is a shorter than t_c and t_c is not minimal. □

The pseudo code of the CTBS algorithm is shown in Listing 5.2. The algorithm uses four data structures: The vertex `root` is the root vertex of the prefix tree, a state `initial` stores the initial state of the TS, a map `v_map` returns vertices that contain a certain state, and a map `t_map` returns a list of transitions with a certain state as the target state.

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

The algorithm adds the transitions belonging to an execution to the `t_map` and calls, for every transition $t = (s_1, a, s_2)$, the function `addTransition` (line 11-13). This function checks for every vertex v with state s_2 (line 17) whether a vertex v_1 with state s_1 and edge (v_1, a, v) (line 19) does satisfy one of the prune rules PR1 and PR2 (line 21), and in this case continues with the next vertex (line 22). Otherwise, both prune rules are not satisfied, s_1 is part of a new minimal suffix and the state s_1 is added to v_1 (line 23). The algorithm then checks whether any other path of a vertex with state s_1 now contains the path of v_1 (line 24) and removes s_1 from such a vertex. If s_1 is an initial state, then a causal trace is found and all other paths in the tree are checked to determine whether they contain the new causal trace (line 26). The transitions of a prefix that reaches state s_1 can already be contained in the transition list `t_map` and need again to be added to the BCT tree. Therefore, the function calls itself recursively for all transitions t' with s_1 as a target state (line 28-29). In order to ensure a clear presentation, we removed several optimization from the pseudo code. Most importantly, in order to optimize performance `addExecution` calls `addTransition` only for transitions not yet in `t_map`. Furthermore, if v_1 in the function `addTransition` already contains the state s_1, then this vertex is ignored and the recursion is not called.

```
1  Vertex root; State initial;
2  Map<State, Set<Vertex>> v_map;
3  Map<State, Set<Transition>> t_map;
4
5  function addExecution(Exection e)
6     IF e.isBad()
7         //add property violating states to root
8         root.addState(e.lastState());
9     initial = e.firstState();
10    //iterate execution transitions
11    FOR Transition t in e
12        t_map.get(t.s2).add(t);
13        addTransition(t.s1, t.act, t.s2);
14
```

```
15 function addTransition(s1, act, s2)
16    //get all vertices with state s2
17    FOR Vertex v2 in v_map(s2)
18       //get father vertex reachable by label act
19       Vertex v1 = v2.getFather(act)
20       //ensure restriction PR1 and PR2
21       IF causalPathShorter(v1) || otherShorter(v1, s1)
22          continue;
23       v1.add(s1);
24       checkOtherLonger(v1, s1); //enforce PR2
25       IF s1 == initial
26          checkAllPaths(v1);     //enforce PR1
27          return;
28       FOR Transition t' in t_map.get(s1)
29          addTransition(t'.s1, t'.act, t'.s2);
```

Listing 5.2: Sketch of Causal Trace Backward Search Algorithm

5.3.3 Correctness of the CTBS algorithm.

The CTBS algorithm returns a BCT tree where every of its vertices that contains the initial state describes a path that represents a causal trace. The CTBS algorithm is correct if the set of causal traces Υ that the BCT tree represents is a causal trace set according to Definition 30. This is the case when Υ is a set of action traces of the transitions system for which the BCT tree was computed, and Υ satisfies conditions TC1 and TC2 from Definition 30.

In Lemma 6, we state that Υ is a set of action traces by a more strict condition that the path of every vertex in the BCT tree to the root describes an action sequence that is a suffix of an action trace of the underlying TS. That is valid, because when a vertex is labeled with an initial state, then the suffix is an action trace of the transition system. Hence, Υ is the set of action traces starting in a vertex with an initial state. Lemma 7 and Lemma 8 ensure that TC1 and TC2 hold for Υ.

Lemma 6. *The path of every vertex in PT to the root is labeled with actions of T such that the action sequence $a_x ... a_y$ is a suffix of an action trace $\langle a_0, \ldots, a_x, \ldots, a_y \rangle$ of T.*

Proof. Assume a BCT tree PT computed by the CTBS algorithm for a TS T with an initial state s_0. Assume a vertex v in PT with an action sequence $a_x, \ldots, a_y$ to the root node. We now show that an execution e in T reaches a state s in v and that e can be extended to an execution that reaches a state in the root vertex. s is added to PT (line 5) by some execution of T with a prefix $s_0 a_0 \ldots a_{x-1} s$ that reaches s. s is added to v in line 23, when an edge labeled with a_x reaches another vertex v_{i+1} in PT and a transition (s, a_x, s_{i+1}) exists in T with s_{i+1} in v_{i+1} (line 17). We append this transition to e and, hence, $s_0 a_0 \ldots a_{x-1} s a_x s_{i+1}$ is an execution of T. For s_{i+1}, also a transition towards another vertex has to exist, and we extend e until e reaches a state s_n in the root vertex. Thus, an execution $s_0 a_0 \ldots s, a_x, s_{i+1}, \ldots, a_y, s_n$ is an

execution of T. This execution has the action trace $\langle a_0 \dots a_x \dots a_y \rangle$. We see that $\langle a_x \dots a_y \rangle$ is a suffix of an action trace in T. □

Lemma 7. *The set of action traces represented by the BCT tree computed by the CTBS algorithm satisfies condition TC1 of Definition 30.*

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

1. t was never added as an action trace to PT. This means that in the sequence of transitions that corresponds to action trace t there is at least one transition (s_i, a_i, s_{i+1}) that is not a transition in PT. That is the case when this transition was never handed over by the state space exploration algorithm. A state space exploration algorithm explores the full state space of an T and every transition of T is contained in at least one execution. Since t contains a transition that is not contained in T, t is not an action trace of T. This contradicts the assumption that t is a bad trace of T.

2. t was deleted by a prune rule because t contains a shorter bad trace t'. In that case, t' is an action trace in Υ. Otherwise, t' is also deleted because of an even shorter bad trace that is contained in PT.

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

Notice that this completeness result holds for the CTBS algorithm also up to the current search depth when it is used with a BFS algorithm for the state space exploration. Assume that the completeness does not hold up to the current search depth. Then there would be a bad trace of length shorter than the current search depth that contains an unexplored transition, which means that the corresponding action trace is not included in the BCT tree PT. This, however, contradicts the property of a BFS that all states up to the current search depth have been explored. This property is beneficial in case we need to bound the search depth in Causality Checking for very large models since it ensures that the causal traces up to the current search depth form a causal trace set, which implies completeness up to the search depth reached.

Lemma 8. *The set of action traces in the BCT tree computed by the CTBS algorithm is minimal according to TC2 of Definition 30.*

Proof. Assume a BCT tree PT computed by the CTBS algorithm for a TS T, and a vertex with an initial state s_0. PT contains an action trace t when a vertex v labeled with s_0 exists and v has a path to the root vertex such that the path edges are labeled with t. The set of action traces in PT is not minimal, when action traces t and t' are in the set with $t' \sqsubset t$. For t, a vertex v with the initial state

s_0 exists, and for t', another vertex v' with the initial state s_0 exists. During the construction of PT, one of the vertices already contains s_0 and with adding s_0 to the other vertex, the corresponding action traces t and t' are compared by prune rule PR1. This leads to two cases. If v already contains s_0, then s_0 is removed from v. Otherwise, if v' already contains s_0, then s_0 is not added to v. In both cases, v does not contain s_0. This contradicts the assumption that t is an initial action trace. □

Lemmas 6 to 8 show that the CTBS algorithm computes a causal trace set Υ. TC1 ensures that Υ is complete. Υ is also sound since Υ is minimal (TC2) and contains only action traces of T as stated by Lemma 6. We now discuss the termination of the algorithm. Notice that a model checker calls the CTBS algorithm for a finite number of executions. Thus, `addTransition` is only called for finitely many transitions and the function returns as the BCT tree contains only finite suffixes because of PR2. Hence, the CTBS algorithm terminates and is correct.

5.3.4 Complexity Considerations.

The *worst-case size of the causal trace set* is bounded by the number of action traces in a TS, since all action traces can be causal traces. A causal trace can be an arbitrary sequence of actions from the action set Act, where the corresponding execution of the TS contains any state at most once. Action traces generated by loops in the TS do not need to be considered in the complexity analysis, as can easily be seen. Assume a causal trace σ_l that reaches a state twice, then the subtrace between reaching the state for the first and the second time corresponds to a loop in the TS. An action trace σ_n not including this loop is shorter and reaches the same end state as σ_l, which means that σ_n is a subtrace of σ_l. Since $\sigma_n \sqsubset \sigma_l$ violates TC2 in Definition 30, σ_l is not a causal trace. In the worst case, the number of action traces in a TS is $|Act|^{n-1}$, where $|Act|$ is the number of actions and n is the number of states $|S|$ in the TS.

The *worst case complexity of the DSPM algorithm* was shown in [102] to be in $\mathcal{O}(|t|^2)$ where $|t|$ is the number of all action traces in a TS. The computational effort is dominated by the comparison of every action trace in t with all causal traces, which in the worst case can be all action traces.

The *worst case complexity of the CTBS algorithm* is also dominated by the number of action trace comparisons. The algorithm processes iteratively the set of transitions of a TS. The number of transitions is at most $|t|$ in the worst case in which all action traces t consist of a single transition. A transition can only be added once to an action trace in the BCT tree. Otherwise the action trace would contain a loop since it would contain the target state of the transition twice. In the worst case a transition is added to all action traces in the BCT tree. When adding a state to a vertex, the prune rules need to compare the action trace of this vertex in the worst case twice with all other action traces. We conclude that the CTBS algorithm has a worst case complexity of $|t| \cdot |t| \cdot 2 \cdot |t| \in \mathcal{O}(|t|^3)$.

In conclusion, under worst case complexity considerations, DSPM scales better than CTBS. The approximation performed by the DSPM algorithm does not affect

the worst case runtime since the action trace set t consists of all action traces, including the action traces not analyzed by DSPM.

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

5.3.5 Computing a Causal Trace Set with Non-Occurrence Events.

As mentioned before, we need to consider non-occurrence (see Definition 29) to compute causes according to Definition 28 for transition systems that deterministically violate a property. By Definition 29, non-occurrence events are additional events in a good trace of which a subtrace is a causal trace. In line with the iterative approach in [102], we compute non-occurrence events by performing a second search in the transition system. In the following, we present an algorithm to compute the good traces of a transition system that contain non-occurrence events. These good traces represent the causal trace set with non-occurrence events. With this causal trace set as an input, the order computation in the following Chapter 6 returns causes according to Definition 28.

The algorithm in Listing 5.3 computes good traces with non-occurrence events and uses the results of the CTBS algorithm in Listing 5.2. The CTBS algorithm computes causal traces Υ, adds every property violating state to the vertex `root` and stores the transition system T since every transition of T is in the map t_map and the initial state `initial` is known. The idea of algorithm in Listing 5.3 is to search the simple action traces of T, and, in case an action trace is a subset of a causal trace in Υ, to add it to a set Υ_N.

The algorithm to compute good traces with non-occurrence events is implemented in the function `computeNonOccurTraces`. Because the search direction starts now in the initial state and not from a property violating state in the root, the look-up map is reversed in line 7, such that it now returns, for a state s_1, the transitions with an action `act` towards a state s_2. We use a stack `st` to search the action traces of T by a depth-first search (DFS) that starts in the initial state `init` (line 8). An element on the stack stores the current state `s`, the action that reaches s and an index of the next transition in $t\_map(s)$, which needs to be explored. While the stack is not empty (line 9), the DFS explores the transitions from the state on the top of the stack towards other states (line 10). The DFS backtracks when either every transition in $t\_map(s)$ is explored, a property violating state is reached or a state is reached that is already on the stack (lines 11 to 13). Otherwise, the transition towards the next state is pushed on `st` (line 14). Notice that the actions on the stack form an action trace of T that is a good trace. Whenever a causal trace is a subtrace of the action trace on the stack (line 15), the function

`addTraceAndSanity` adds it to a set Υ_N. In Υ_N, the function stores only minimal good traces, in other words, a sanity check ensures that no action trace is a subtrace of another action trace in Υ_N. After the termination of the DFS, Υ_N contains only action traces with actions that are either part of the causal trace or non-occurrence actions as stated in Theorem 7.

```
1  Set<ActionTraces> Υ; //causal traces of CTBS algorithm
2  Set<ActionTraces> Υ_N; //result set
3  Vertex root; //property violating states
4  State init; //initial state
5  Map<State, Set<Transition>> t_map;
6  function computeNonOccurTraces()
7     t_map = reverse(t_map);
8     stack<State s, Action, Int idx> st.push(init, null, 0);
9     while !s.empty()
10       Transition t = t_map(st.peek.state)[st.peek.idx++]
11       if (t==null) or (t.s2 in root) or (t.s2 in st)
12          //no transition or property violation or loop
13          st.pop(); continue;
14       st.push(t.s2, t.act, 0);
15       if st.getTrace().containsCausalTrace(Υ)
16          //adds trace and removes non minimal traces
17          Υ_N.addTraceAndSanity(s.getTrace());
18          st.pop(); continue;
```

Listing 5.3: Sketch of Non-Occurrence Search Algorithm

The algorithm in Listing 5.3 can compute good traces with a fixed number of non-occurrence events. In line with the implementation in [102], we consider only non-occurrence action traces with a single non-occurrence event. In addition, we observed that a prefix of an action trace that contains non-occurrence events is similar to a prefix of the causal trace that the action trace contains. A prefix of action trace differs from a prefix of the causal trace only in the non-occurrence events. For an efficient implementation of the causal trace comparison in line 15, we store the causal traces by a prefix tree ct and synchronize the search of the DFS with a search of the causal trace in ct as follows. The stack and a causal trace in ct start in the initial state, hence, we add the root vertex of ct with the initial state to the initial element on the stack. When we add an action a to the stack, we also search for an edge in ct labeled with a that starts in this vertex. In case a transition exists, the action trace on the stack and in ct describe the same prefix, and we add the vertex that this prefix reaches in ct to the new top-level element on the stack. Otherwise, no transition exists and a potential non-occurrence event is found. When this happens for the first time, we add the vertex added to the previous top-level element to the current top-level element on the stack, or backtrack otherwise. By this approach, only one DFS is necessary to search the T and the ct, and at most one non-occurrence element is on the stack. This optimizes the algorithm since the DFS only searches the action traces of T that differs from a prefix of a causal trace in at most one event.

Correctness. The algorithm in Listing 5.3 terminates because it searches the simple action traces of a transition system and only a finite number of simple action traces exist in a transition system. In Theorem 7, we show that the algorithm in Listing 5.3 computes the action traces that contain only non-occurrence events or the actions of a cause according to Definition 29.

Theorem 7 (Soundness)**.** *Assume a set Υ_N computed by the algorithm in Listing 5.3 for a transition system and a regular safety property. The algorithm in Listing 5.3 is* sound *if every action trace in Υ_N is an action trace $\hat{\sigma}$ according to Definition 29.*

Proof. Assume the causal trace Υ for a transition system T and a regular safety property φ, and a set Υ_N computed by the algorithm in Listing 5.3. An action trace σ in Υ_N is an action trace $\hat{\sigma}$ according to Definition 29 when σ satisfies a cause and φ, is minimal since no subtrace does this.

Assume an arbitrary action trace σ in Υ_N and the causal trace σ_c with $\sigma_c \sqsubset \sigma$. σ_c has to exist because otherwise σ would not be added to Υ_N (lines 15 to 17). σ_c is a causal trace for a cause ψ with an action set $\mathcal{A}_\psi$. ψ does not satisfy CC2(2) since $\sigma \vDash \psi$ and $\sigma \vDash \varphi$. σ is also minimal since no other trace in Υ_N is a subtrace of σ ensured by the sanity check of the function `addTraceAndSanity` (line 17). Thus, σ is an action trace $\hat{\sigma}$ as assumed in Definition 29. □

The DFS in Listing 5.3 explores every simple action trace of a TS and compares it with the causal traces. Thus, the algorithm is complete as stated by Theorem 8 when every action trace of the TS that is an action trace $\hat{\sigma}$ according to Definition 29 is in Υ_N.

Theorem 8 (Completeness)**.** *Assume a set Υ_N computed by the algorithm in Listing 5.3 for a transition system T and regular safety property. The algorithm in Listing 5.3 is* complete *if every action trace of T that is an action trace $\hat{\sigma}$ according to Definition 29 is in Υ_N.*

Proof. Assume the causal trace Υ for a transition system T and a regular safety property φ, and a set Υ_N computed by the algorithm in Listing 5.3. We assume a simple action trace σ of T that is an action trace $\hat{\sigma}$ according to Definition 29 but not in Υ_N. Because σ is according to Definition 29, σ is an action trace of T, σ satisfies a cause Ψ and φ. The algorithm in Listing 5.3 does not add σ to Υ_N in the following cases.

σ is not in Υ_N, when a state in σ violates φ (line 11-13). When a state in σ violates φ, σ violates φ which contradicts our assumption that σ satisfies φ.

σ is not in Υ_N when σ is never on the stack of the DFS in Listing 5.3. Notice, that σ is simple and satisfies φ. σ is never on the stack when a transition t in σ is not in *t_map* (c.f. lines 10). This missing t contradicts that the CTBS is complete as stated in Lemma 7.

Finally, $\sigma \notin \Upsilon_N$ holds when no causal trace in Υ is a subtrace of σ. We assumed σ satisfies a cause ψ, hence, for a causal trace σ_c of ψ holds $\sigma_c \sqsubseteq \sigma$. Since the CTBS algorithm is complete by Lemma 7, σ_c is also in Υ and compared with σ in line 15. This contradicts that σ_c is not in Υ.

In every case, the assumed σ cannot exist and, thus, the algorithm in Listing 5.3 is complete. □

The algorithm in Listing 5.3 terminates because it search a finite number of simple action traces. The algorithm in Listing 5.3 is correct since it terminates, is sound by Theorem 7 and complete by Theorem 8.

5.4 Case Study and Experimental Evaluation

We now compare the DSPM and CTSB algorithms by analyzing several models of different size taken from [102] and the autonomous driving models presented in Chapter 7. We implemented four variants of the considered algorithms:

(1) the CTBS algorithm based on a BFS state space exploration,

(2) the CTBS algorithm based on a DFS state space exploration,

(3) a modified version of the DSPM algorithm, and

(4) to obtain a baseline, an algorithm that ignores all duplicate states, i.e., behaves like a standard BFS when encountering a duplicate state.

These implementations are integrated in the `QuantUM` tool and use a modified version of the model checker SpinJa [41] for the state space exploration. We modified the above described DSPM algorithm in order to address its insufficiencies with respect to treating multiple duplicates encountered during state space exploration. This is accomplished by computing all minimal counterexamples by repeatedly iterating through all duplicate states stored in a list maintained by DSPM until no new counterexample is generated.

All analyses were performed on a computer with an E5-2697 CPU with 24 cores (2.7GHz), 785GB of RAM and a 64 bit Linux operating system. The analysis algorithms were mapped to two threads: one thread performed the state space exploration and another executed the algorithm for the causal trace set computation. The state space exploration was based on a BFS as long as nothing else is stated. A timeout for the experiments was set to two hours.

Table 5.1 shows the results of the CTBS algorithm. The input model is characterized by the number of states that were explored, the number of transitions traversed and the maximal search depth. For the larger models `FFU Star`, `FFU ECU`, `ASR ch1` and `ASR Reduced`, we limited the search depth to 60. In case of a timeout, the current search depth reached by the CTBS algorithm is given in column *Depth*. For all of the four larger models, the search depth limit was reached by the state space exploration when the timeout occurred. We indicate time values using the format (hours:minutes:seconds). The column $\#CT$ gives the size of the causal trace set that was computed, $\#C$ gives the number of causes that were detected, and $\text{Time}_{\text{Last}}$ indicates the period of time after starting the experiment when the last causal trace was found. The computation time consumed by the analysis is given in the Time_{A} column, and the consumed memory in the column *Memory*. Memory is given in MB. Table 5.2 shows some experimental results for the algorithm variants 4), 3) and 2).

Model	States	Transitions	Depth	#CT	#C	$\text{Time}_{\text{Last}}$	Time_{A}	Memory
Railroad	143	373	37	62	4	<00:01	00:04	51
Airbag	3,456	14,257	35	484	5	<00:01	07:09	62
Odometer	4,032	19,624	55	13	3	<00:01	00:09	70
FFU Star	199,206	921,463	37/60	458	16	<00:02	timeout	697
FFU ECU	226,688	1,147,834	31/60	509	19	<00:02	timeout	863
ASR ch1	680,897	3,745,635	37/60	67200	2	02:00:05	timeout	3,747
ASR Reduced	14,222,115	90,775,575	39/60	76428	2	01:45:58	timeout	91,814

Table 5.1: Experimental Results for the CTBS Algorithm (Variant 1).

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

Table 5.2: Experimental Results for the Algorithm Variants 4), 3) and 2).

In order to compare the algorithm variants qualitatively, we analyzed the railroad crossing example taken from [102] with the CTBS variant 1) using the QuantUM tool. We obtained the fault tree depicted in Figure 5.3. The fault tree has four subtrees that represent the causes `Cause 1` ... `Cause 4`. A cause is created by combining all causal traces with the same set of actions. The number of causal traces that a cause contains is depicted next to the cause name, for instance, `Cause 1` contains 20 causal traces. These causal traces differ only in the action order of the 5 actions in `Cause 1`. The depicted order of the actions from top to the bottom is the action order in one of the causal traces. Notice that the type of fault tree that we use is a dynamic fault tree, which means that the occurrence order of the actions on one of its and-branches is assumed to be arbitrarily ordered. In effect, the causal traces of a cause contain a partial order of the depicted events that cannot yet be directly depicted in a fault tree. We present an algorithm to compute partial order in Chapter 6, and omit these order constraints here.

After the causal traces are computed, non-occurrence is analyzed for models that deterministically violates a property. As expected the algorithm in Listing 5.3 detected that the models `FFU ECU` and `FFU Star` non-deterministically violate their properties. We created deterministic versions `FFU ECU (det)` and `FFU Star (det)` of these models by deleting non-deterministic transitions and rerun the algorithm. The results of the algorithm in Listing 5.3 are given in Table 5.4. The column $\#C$ gives the number of causes after the non-occurrence computation, $\text{Trace}_{\text{Non}}$ presents the number of action traces with a non-occurrence action, Time_{Non} states the time between the start of the algorithm in Listing 5.3 and its termination and Mem_{Non} indicates the maximal increase of the memory consumption while the algorithm runs.

Model	#C	$\text{Trace}_{\text{Non}}$	Time_{Non}	Mem_{Non}
Railroad	5	116	<00:00:01	0.77
Airbag	5	0	00:00:01	8.46
Odometer	3	3	00:00:01	<0.01
FFU ECU (det)	28	656	00:00:18	7.18
FFU Star (det)	25	632	00:00:19	7.97
ASR Ch1	2	0	00:01:50	3,072.74
ASR Reduced	2	0	00:16:41	3,696.03

Table 5.3: Experimental Results for Non-Occurrence Computation.

5.4.1 Quantitative Result Interpretation.

The algorithm variant 4) defines a base line for the quantitative performance of the other algorithm variants when analyzing the different models. The highest analysis time with this algorithm is observed for the model `ASR Reduced` with a value of 57:57. The algorithm variants 1), 2) and 3) have a much higher computation time demand. As opposed to the baseline, these algorithms experienced a timeout for the models `FFU Star`, `FFU ECU`, `ASR ch1` and `ASR Reduced`. We can conclude that the penalty for processing duplicate states completely is a significant increase in computation time. Algorithm variant 3), which corresponds to the modified DSPM algorithm, experiences a timeout for all models, whereas CTBS in both variants 1) and 2) is able to analyze some models without timeout. This points to a performance advantage of CTBS over DPSM.

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

The CTBS algorithm based on a BFS (variant 1) performs differently compared to the DFS based version (variant 2) in terms of runtime and causal results. For the `Airbag` model, variant 1) requires 7:09 which is substantially less than 39:46 for variant 2). A DFS based algorithm first searches the depth and checks many orderings of non causal traces before shorter causal traces are found. The CTBS

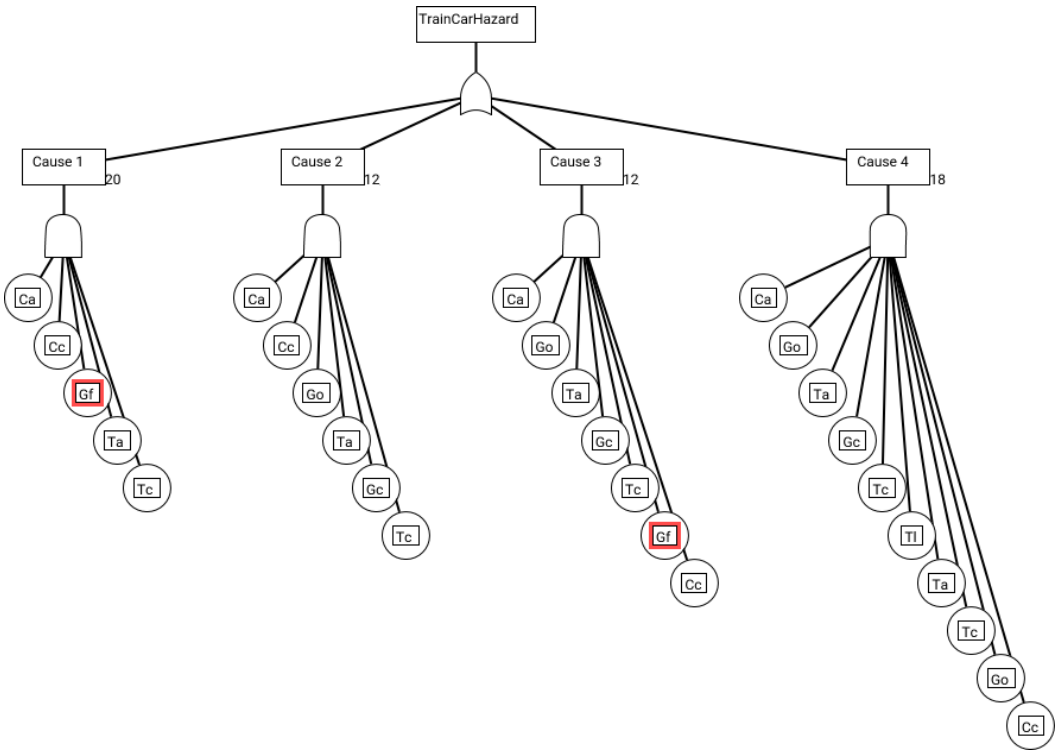


Figure 5.3: Fault Tree for Railroad Crossing Using CTBS variant 1)

algorithm based on DFS scales worse than based on BFS and this result goes in line with previous results in [104]. Variant 2), however, found more causes than variant 1), for the `FFU Star`, `ASR ch1` and `ASR reduced` model. We had a closer look at the fault trees and detected that several causal traces returned by variant 2) are actually not causal. As expected, the DFS cannot ensure that a causal trace is causal until the algorithm terminates. We conclude that variant 2) is not sound when the analysis is aborted before completion, for instance due to a timeout. For larger models, the BFS should be used to obtain correct results.

The maximal time that the algorithm in Listing 5.3 needs to compute non-occurrence events is 16 minutes and 41 seconds with memory consumption of 3,696.03MB for the `ASR Reduced` model. This resource consumption is significantly less than the resource consumption of the CTBS algorithm that reaches the 2h timeout and uses 91,814 MB of memory. The time efficiency is reasonable since the algorithm in Listing 5.3 explores only action traces that differ from a causal trace in at most one event. Notice that the before executed CTBS algorithm already stored all transitions of an analyzed model. This explains the memory efficiency of the algorithm.

5.4.2 Qualitative Result Interpretation.

We had a detailed look at the causal traces with non-occurrence events that the algorithm in Listing 5.3 computed and they are reasonable. We further discuss

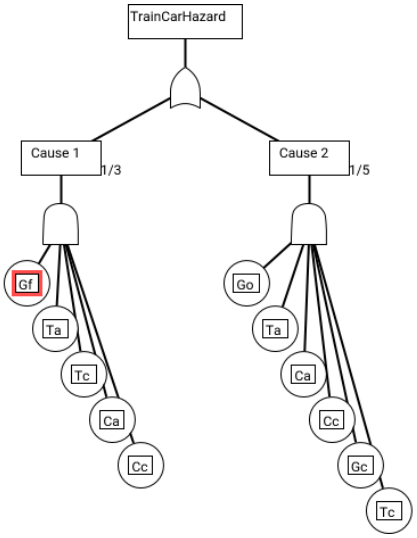


Figure 5.4: Fault Tree for Railroad Crossing Using CTBS variant 4)

the non-occurrence results in Chapter 6 as in that chapter causal traces with non-occurrence events and the event orders that motivate cause splits are visualized.

We refer to the railroad crossing in Example 1 to illustrate some qualitative differences between variant 1) and variant 4), the algorithm defining the baseline in terms of quantitative performance.

The fault tree in Figure 5.4 computed for the railroad crossing example by variant 4) is missing the causes labeled `Cause 3` and `Cause 4`, which were on the other hand computed by variant 1), see the fault tree in Figure 5.3. The events of `Cause 1` are a subset of the events in `Cause 3`, but the action Gf happens after the action Tc. The events of `Cause 2` are contained in `Cause 4` but the action Cc happens after the action Gc. When Gf happens before Tc, the gate stays falsely open. When Gf happens after Tc then the gate falsely opens itself. From `Cause 2` we can infer that an accident happens because the car is not leaving the crossing. From `Cause 4` computed by CTBS in variant 1) we conclude that an accident can additionally happen because the train is not leaving the crossing. This exemplifies that CTBS in variant 1) can compute more information regarding the cause of a property violation in the system than the baseline algorithm variant 4). This is due to the completeness of variant 1) as opposed to the incompleteness of variant 4).

5.5 Conclusion

We have addressed the complete computation of causal trace sets in Causality Checking. The complete computation of the causal trace sets is essential in the analysis of safety-critical systems in order to ensure that all causal factors will be identified by the analysis. The CTBS algorithm that we propose addresses the problem of a complete and sound construction of action traces that belong to this set when the state space traversal encounters multiple duplicate states during the

search. We contrast the CTBS algorithm with the DSPM algorithm, which is the current basis for implementations of Causality Checking. The variant of DSPM originally implemented in QuantUM only performs an incomplete under-approximative handling of the construction of traces when encountering duplicate states. In contrast, CTBS handles the encounter of duplicate states properly and completely and manages to establish complete causal sets. In particular, CTBS outperforms a variant of DSPM modified to accomplish a naive correction of the under-approximation.

The results also showed that it is beneficial to use CTBS with BFS instead of DFS. With BFS, we can ensure that the results are correct up to the current search depth. This makes it possible to analyze even large models where the implementation of Causality Checking reaches its limits.

Chapter 6

Analyzing the Strict Partial Order in Action Traces

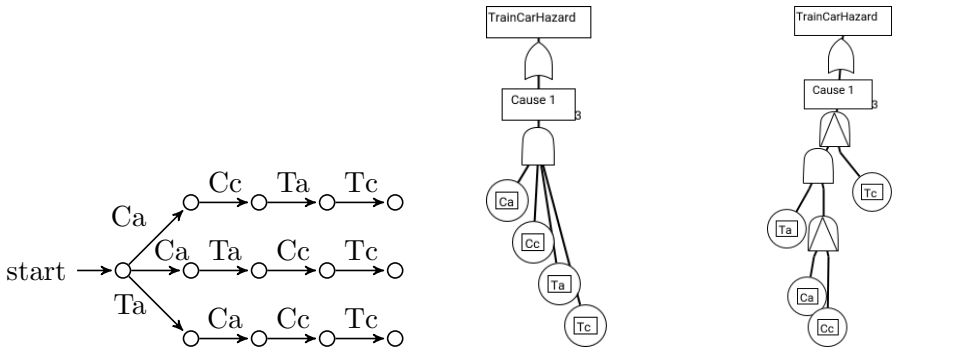
The content of this part is based on the publication [88].

Contents

6.1 Introduction

QuantUM in [102] implements Causality Checking and depicts the actions of a cause by a fault tree without indicating the order of the contained actions. We now propose an order analysis to compute the strict partial order of the actions in a cause. We illustrate the order analysis by applying it to a simplified model of the railroad crossing in Example 1. In this simplified model, the crossing has no gate and, hence, is unguarded. We will refer to this unguarded railroad crossing model as a running example throughout the chapter. In this model, a train approaches a crossing (Ta), then enters the crossing (Tc) and leaves the crossing (Tl). A car also approaches at the crossing (Ca), then enters the crossing (Cc) and leaves the crossing (Cl). The car will not enter the crossing when the train is already in the crossing. A property violation in this system occurs when a state can be entered in which both the train and the car are in the crossing at the same time, which has the potential to lead to a fatal accident. We, therefore, state the property that such a state can not be reached.

Remember, Causality Checking computes a causal trace set (c.f. Definition 30) for the violation of such a reachability property in case it is violated in the model. By CC1 in Definition 22 and in Definition 28, a cause describes a set of minimal bad traces over the same multi-set of actions. Thus, we split the causal trace set in causes where every action trace in a cause is formed over the same multi-set of actions, and only vary in the order in which these actions occur. For the unguarded railroad crossing example, Causality Checking computes just one cause over the set {Ta, Tc, Ca, Cc} of actions. It contains 3 causal traces, depicted in Figure 6.1(a). Notice that for systems of realistic size, a cause may contain a much higher number



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

Figure 6.1: Unguarded Railroad Example

of action traces. For reasons of convenience and since QuantUM is primarily used in the area of safety-critical system analysis, causes are depicted as fault trees. The fault tree in Figure 6.1(b) depicts the cause computed for the unguarded railroad crossing example. The top-level event *TrainCarHazard* is valid when one of the causes is valid. Thus, an or-gate is connected to the single cause *Cause 1*. A cause is valid when every contained basic event Ca, Cc, Ta and Tc occurred in the depicted order, which means that an action trace is obtained from the system that contains these events in this order.

The basic events can occur in the model in different orders. These basic events are combined by an and-gate that requires all of them to occur without imposing a particular order on the occurrence. This fault tree returned by QuantUM does not currently depict these different action orders, even though the ordering information is contained in the set of action traces that form the cause. Assume, in the example above, that the train enters the crossing before the car, then the car will never enter the crossing, as per the model definition, and the property violating state will not be reached.

We propose an analysis that computes the strict partial orders of actions in action traces and depicts the computed order in a fault tree. In order to depict the action orders, we use the ordered-and-gate into the fault tree notation introduced in [103]. It is depicted as an and-gate labeled with a triangle. It is satisfied when the actions connected to the gate occur from left to right. For the running example, the analysis results in the fault tree depicted in Figure 6.1(c), which represents all orders of the action traces given in Figure 6.1(a) and which corresponds to the causality criteria in Definition 22 of Causality Checking. The order in the fault tree depicts, for instance, that Tc occurs always after the other actions, and the action Ta is independent of action Ca and action Cc. In this chapter, we present algorithms that compute and depict the order of actions in the set of causal traces belonging to a cause. We also implement these algorithms in QuantUM.

Structure. In Section 6.2, we present algorithms to compute the strict partial order for the causal traces of a cause. We evaluate an implementation of the algorithms

in Section 6.3. In Section 6.4, we draw conclusions.

6.2 Algorithm to Analyze Action Orders in Action Traces

In this section, we present an algorithm that we refer to as Algorithm 6.1, which is designed to compute the strict partial order of the actions in a set of causal traces. We also define an algorithm called Algorithm 6.2, which translates this strict partial order into a fault tree.

The input to Algorithm 6.1 is an action trace class defining the causal traces of a cause. For instance, in the railroad example the causal traces in Figure 6.1(a) are the action trace class of the cause computed for the railroad model. This action trace class is built over the action set {Ca, Cc, Ta, Tc}. Algorithm 6.1 computes the strict partial order of the actions in an action trace class and stores it in a DAG which is accomplished in the following way. The strict partial order $a \prec b$ holds for an action trace in which an action a occurs before an action b. In the railroad model, for instance, Tc occurs in every action trace after Cc and Ca. Thus, Cc $\prec$ Tc and Ca $\prec$ Cc holds in every action trace. Since Cc $\prec$ Tc holds in every action trace in Figure 6.1(a), we deduce that Tc depends on Cc occurring first in order to reach a property violation. In the same way, Cc also depends on Ca and Tc depends on Ca. In the DAG for the railroad example, the algorithm only needs to store the information that Tc depends on Cc and that Cc depends on action Ca because Tc also transitively depends on Ca. We define the direct dependency relation in Definition 31 that removes transitive dependency relations.

Definition 31 (Direct Dependency Relation $\hat{\prec}$)**.** *An action b directly depends on action a, written as $a \hat{\prec} b$, in an action trace class C with an action set A when $a \prec b$ holds and $\neg\exists a' \in A.a \prec a' \wedge a' \prec b$.*

For instance, in the running example, the direct dependency relations Ca $\hat{\prec}$ Cc and Cc $\hat{\prec}$ Tc hold and imply the dependency relation Ca $\prec$ Tc but Ca $\hat{\not\prec}$ Tc because Cc has a direct dependency with Ca and Tc.

The DAG in which the algorithm stores the dependencies is a DDAG G defined in Definition 32. A DDAG has no superfluous edge e that can be implied by transitivity, formally $(E \backslash e)^* = E^*$, and stores this transitive reduction of the strict partial order.

Definition 32 (Dependency DAG (DDAG))**.** *A dependency DAG (DDAG) is a DAG (V, E) that stores a strict partial order $\prec$ over a set A with $V = A$ and $(a, b) \in E$ for any actions $a, b \in A$ where $a \hat{\prec} b$ holds.*

We say an action trace σ satisfies a DDAG G when σ is consistent with the strict partial order $\prec$ stored in G.

G is the input for Algorithm 6.2, which computes a causal tree as defined in Definition 33. The causal tree represents the strict partial order in G and also represents implicit independencies in G. In the context of Causality Checking, a causal tree is the part of a fault tree that represents a single cause. A causal tree consists of basic events that represent the actions in a cause, ordered-and-gates where a connected event on the right side depends on every event on the left

side, and and-gates where the connected events are independent. The fault tree in Figure 6.1(c) depicts the basic events Ca, Cc, Ta and Tc. In the fault tree, an ordered-and-gate specifies that Cc depends on Ca to occur first, and a regular and-gate specifies that Ta and Cc are independent.

Definition 33 (Causal Tree)**.** *A causal tree CT is a connected DAG where a vertex v is either a basic event for an action, or it is a gate. Any vertex v can have an edge (v, g) to a gate g. A gate g is either an ordered-and-gate, where the vertices $v_0...v_j$ with edges (v_i, g) are ordered by increasing index i from left to right, or an and-gate that does not impose an order of the vertices attached to it.*

A causal tree $CT = (V, E)$ describes a partial order $\prec_T$ with an event set $\mathcal{A} = V$ where (v_i, v_j) is in $\prec_T$ for every two different vertices v_i and v_j in V that have walks $v_i...v_l g$ and $v_j...v_k g$ to an ordered-and-gate g in (V, E) and $l < k$ for g. Formally, an action trace σ satisfies (V, E) when σ is consistent with $\prec_T$.

In order to derive a fault tree, we compute a causal tree for every cause and combine the obtained causal trees with an or-gate. For the railroad example, the result of these computations is the fault tree depicted in Figure 6.1(c). It contains one causal tree which is the subgraph below and including the ordered-and-gate, denoted by the and-gate symbol labeled with a triangle.

Algorithm 6.1 computes the strict partial order of the actions in an action trace class and stores the resulting strict partial order relation in a DDAG. The functions given in Listing 6.1 compute a DDAG with the strict partial order for a given action trace class `C` built from an action set A. Let $i_\sigma(a)$ be a function that returns the index of an action a in an action trace σ, and the function $\sigma[i]$ return the action in σ at index i. The function `createPreconditionMap` preprocesses the action traces of an action trace class and stores in a map `aM` for every action b in A, the set of actions that occurred in an action trace directly before b. The function iterates in lines 3 to 5 through every action $\sigma[i]$ in every action trace σ and adds the action $\sigma[i-1]$ occurring before $\sigma[i]$ to the set of $\sigma[i]$ in aM.

The function `createDAG` obtains the map `aM` as an input and computes a DDAG representing the strict partial order of a given action trace class `C`. The algorithm uses `aM` as an input in line 8 to create a dependency matrix `m` of size $|A| \times |A|$. For any actions a and b, an entry (a, b) in `m` is true when $i_\sigma(a) < i_\sigma(b)$ holds in an action trace σ of C. Hence, (a, b) is true when an action a is in `aM[b]`. Next, the algorithm ensures that the properties of a strict partial order hold for the relation stored in `m`. In line 10, the algorithm computes the transitive closure of `m` and stores it in `m`. In line 12, the algorithm removes symmetries in `m` by setting (a, b) and (b, a) to false since, as we argue in Section 4.3, symmetrically ordered actions cannot be dependent on each other. In line 13, the algorithm removes reflexive transitions when for an action a, the relation $a \prec a$ holds.

```
1 Map<Action, Set<Action>> aM;
2 function createPreconditionMap(Set<ActionTrace> C)
3   for ActionTrace σ in C
4     for i: 1 ... σ.length - 1
5           aM.get(σ[i]).add(σ[i-1]);
```

```
6
7  function createDAG(Map<Action, List<Action>> aM)
8    Matrix m = createDependencyMatrix(aM);
9    //transitive closure: a1 < a2 && a2 < a3 => a1<a3
10   m = ensureTransitivity(m);
11   //antisymmetric: a1<a2 && a2<a1 => independent(a1, a2)
12   m = ensureAntisymmetricity(m);
13   m = removeReflexivity(m);
14   m = removeTransitiveDependencies(m);
15   return getDAG(m);
```

Listing 6.1: Pseudocode of Algorithm 6.1 to Compute DAG.

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

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

Algorithm 6.2 uses the DDAG `G` returned by function `getDAG` as an input and computes a causal tree. In lines 4 to 6 in Listing 6.2, the algorithm first iterates through every action `p` in `G` where `p` is a precondition of another action `a`. It stores this property of p using a Boolean variable `Used` for `p` in a map `m`. In lines 7 to 9, we search for every action that is not a precondition of another action. These actions are independent of any other action. For every independent action `a`, we call the recursive function `createTree` in line 10 in order to create a tree that represents the dependencies of `a`. The function `createTree` creates a tree for the dependencies of an action `a`. In line 14, the algorithm checks whether the tree of an action `a` was previously created. In case, this tree for `a` is stored in `m`, the function `createTree` returns this tree. Otherwise, the algorithm creates the tree for `a` and stores it in variable `t`. The algorithm first creates a basic event `e` for `a` in line 17. In line 18, the algorithm gets the set `pL` of actions on which `a` depends. In case `a` depends on no other action, `e` is the tree with the dependencies of `a` and the algorithm stores `e` in `t`. In case `pL` contains only a single action stored in `pL[0]`, the algorithm creates the tree `t'` with the dependencies for the action in `pL[0]` in line 22. In case `pL` has several actions, the algorithm creates a tree for every action in `pL` in line 24 and combines these trees with an and-gate `t'`. `t'` depicts the precondition actions of `a`, thus, the algorithm combines `t'` and `a` in line 25 in this sequence with an ordered-and-gate. This ordered-and-gate is stored in `t`. `t` is stored in `m` for the action a in line 26 and in line 27 returned by the function.

```
1  Map<Action, (Tree, Used)> m;
2  Set<Tree> indep;
3  function createCausalTree(DDAG G)
```

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

Listing 6.2: Pseudocode of Algorithm 6.2 to Compute Causal Tree.

The function `createTree` is called in line 10 for every independent action. The trees of these actions are stored in a set `indep`. After all trees are created, they are combined by an and-gate in line 11 and this and-gate is the causal tree that we wanted to compute.

It is possible to optimize the algorithm in the following way. An ordered-and-gate in the causal tree can be connected to another ordered-and-gate. For instance, when a_1 occurs before a_2 and a_2 occurs before a_3 then the presented algorithm creates two ordered-and-gates instead of one with all three actions. The implementation of lines 22 and 25 in Listing 6.2 combines several ordered-and-gates to a single one when possible and returns it.

6.2.1 Correctness

We now prove that the presented algorithms to compute a causal tree that depicts a strict partial order for an action trace class C is compelete and sound, and thereby partially correct.

A DDAG G computed by Algorithm 6.1 is complete according to Definition 34 when every action trace in C corresponds to a valid ordering of the actions according to the dependencies stored in G.

Definition 34 (Completeness DDAG Construction)**.** *Assume a DDAG G computed for an action trace class C. G is complete when any action trace $\sigma \in C$ is an action trace satisfying G.*

Theorem 9 (Completeness of Algorithm 6.1)**.** *Algorithm 6.1 computes a complete DDAG according to Definition 34.*

Proof. Assume a DDAG G computed by Algorithm 6.1 for an action trace class C and an action trace σ in C that is not satisfying G. Since σ is not satisfying G two actions a and b exist that satisfy $i_\sigma(b) < i_\sigma(a)$ but in G the dependency relation $a \prec b$ holds. Since $a \prec b$ holds in G by construction of G another trace σ' in C exists that satisfies $i_{\sigma'}(a) < i_{\sigma'}(b)$. For σ and σ', Algorithm 6.1 would store the relations $i_\sigma(b) < i_\sigma(a)$ and $i_{\sigma'}(a) < i_{\sigma'}(b)$ in matrix m (line 8) and removes them (line 12) afterwards since these relations contradict antisymmetricity. Thus, either $a \prec b$ cannot hold in G or $\sigma \notin C$. Both cases contradict our assumptions. □

A DDAG G computed by Algorithm 6.1 could be considered sound when any action trace that satisfies G is in C. However, as we shall see, this definition of soundness is too strict. Assume a set with two action traces $\langle a, b, c\rangle$ and $\langle c, a, b\rangle$. Then, Algorithm 6.1 computes a strict partial order $\{a \prec b\}$ over the event set $\{a, b, c\}$. This strict partial order allows the action trace $\sigma_3 = \langle a, c, b\rangle$ but σ_3 is not in the original action trace class. This observation was considered further in [138]. For G, we therefore use a different soundness criterium based on pairs of actions. Notice that in an action trace that satisfies G, only the order of independent actions can be changed while preserving its satisfaction of G. As mentioned above, two actions a and b are independent when neither $a \prec b$ nor $b \prec a$ holds in G. $a \prec b$ does not hold when an action trace σ with $i_\sigma(b) < i_\sigma(a)$ exists, and $b \prec a$ does not hold when an action trace σ' with $i_{\sigma'}(a) < i_{\sigma'}(b)$ exists. G is sound according to Definition 35 when for any two independent action in G the action traces σ and σ' exist.

Definition 35 (Soundness DDAG Construction)**.** *Assume a DDAG G computed for an action trace class C. G is sound when for any two independent action a and b in G, an action trace $\sigma \in C$ satisfying $i_\sigma(b) < i_\sigma(a)$ exists and another action trace $\sigma' \in C$ satisfying $i_{\sigma'}(a) < i_{\sigma'}(b)$ exists.*

Theorem 10 (Soundness of the Algorithm 6.1)**.** *Algorithm 6.1 computes a sound DDAG according to Definition 35.*

Proof. Assume a DDAG G computed by Algorithm 6.1 for an action trace class C and two actions a and b that are independent in G and $a \neq b$. Partial orders in G are represented by edges. Hence, two actions are independent in G when neither a walk $a...b$ nor a walk $b..a$ exists. By construction of G, a walk $a...b$ does not exist when a trace σ with $i_\sigma(b) < i_\sigma(a)$ exists and a walk $b...a$ does not exist when a trace σ' with $i_{\sigma'}(a) < i_{\sigma'}(b)$ exists. We now show by contradiction that the action traces σ and σ' are in C.

In a first case, we assume that no action trace σ exists that satisfies $i_\sigma(b) < i_\sigma(a)$. Thus, the relation $a \prec b$ is not removed in line 12 in Listing 6.1. In this case, either $a \hat{\prec} b$ and the algorithm creates an edge (a, b) (line 15) or actions a_1, ..., a_n with $a \prec a_1 \prec ... \prec a_n \prec b$ exists and the algorithm creates edges $(a, a_1)(a_1, a_2)\dots(a_n, b)$ in G. Both, the single edge and the sequence of edges represent a walk $a...b$. This walk contradicts the assumption that a and b are independent.

In a second case, we assume that no action trace σ' exists that satisfies $i_{\sigma'}(a) < i_{\sigma'}(b)$. This case is equivalent to the first case since a and b are only substituted with another. Thus, the reasoning that σ' has to exist is similar to the argumentation for σ.

We see that every case contradicts its assumption. Thus, when a and b are independent then σ and σ' have to exist. □

We now discuss whether Algorithm 6.1 terminates. Algorithm 6.1 iterates over actions and their relations. Since the number of action traces in C is finite, the actions and the action relation are also finite. We conclude that Algorithm 6.1 terminates.

Assume that Algorithm 6.2 computes a causal tree CT for a DDAG G. We abuse notation since we refer in the following to an action a that is represented in DDAG as a vertex and in a causal tree as a basic event. Algorithm 6.2 is sound when for any two actions a and b where $a \prec b$ holds in CT, $a \prec b$ holds in G, and the algorithm is complete when $a \prec b$ holds in G then $a \prec b$ holds in CT. Remember that action b depends on a does not imply that b directly depends on a, formally $\neg\forall a, b.\ a \prec b \Rightarrow a \hat{\prec} b$. In G, $a \prec b$ holds when a walk $a \dots b$ exists. In CT, the dependencies of actions are depicted by ordered-and-gates. $a \prec b$ holds in CT when there exists an ordered-and-gate g_b with edges (v_x, g_b) and (b, g_b), where $v_x \prec b$, and a walk $a...v_x g_b$. Definition 36 ensures that an action b depends on an action a in G iff b depends on a in CT.

Definition 36 (Correctness of Causal Tree Construction)**.** *Assume a causal tree CT computed for a DDAG G with an action set $\mathcal{A}$. CT is sound if for any two actions $a, b \in \mathcal{A}$ where $a \prec b$ holds in CT, $a \prec b$ also holds in G. CT is complete if for any two actions $a, b \in \mathcal{A}$ where $a \prec b$ holds in G, $a \prec b$ also holds in CT. CT is partially correct if it is sound and complete.*

Theorem 11 (Correctness of Algorithm 6.2)**.** *Algorithm 6.2 computes a correct causal tree according to Definition 36.*

Proof. Assume a causal tree CT computed by the Algorithm 6.2 for a DDAG G, and two actions a and b in G. In a first case $\Rightarrow$, we assume that $a \prec b$ holds in G and will show that $a \prec b$ holds in CT, and in a second case $\Leftarrow$, we assume that $a \prec b$ holds in CT and will show that $a \prec b$ holds in G. In line 25 of Listing 6.2, an ordered-and-gate g_b is created for b when b directly depends on at least one other action in G. Thus, when b depends on another action, g_b exists and when g_b exists, b depends on another action. By construction of g_b, b is its most right vertex and so for any walk $a..v_x g_b$ in CT, $v_x \prec b$ holds.

- $\Rightarrow$ We assume that $a \prec b$ holds in G, and we now construct a walk $a...g_b$ in CT to show that $a \prec b$ also holds in G. Because $a \prec b$ holds in G, a walk $a_0...a_n$ with $a_0 = a$ and $a_n = b$ in G has to exist. This walk witnesses that every action a_i with $0 < i \leq n$ has a precondition. Thus, Algorithm 6.2 creates an ordered-and-gate g_i for every a_i with $i \geq 1$ (line 25 in Listing 6.2), and for $i > 1$ either an edge (g_{i-1}, g_i) when a_i has a single precondition (line 22), or creates an and-gate g'_i and the edges (g_{i-1}, g'_i) and (g'_i, g_i) (line 24). We see a

walk $g_1...g_n$ has to exist in CT. Action a can also have a precondition then similar to the other actions a walk $ag_0g_1...g_n$ exists in CT. Otherwise, a has no precondition (line 19) and a walk $ag_1...g_n$ exists. Since $g_n = g_b$, both walks $ag_0g_1...g_n$ and $ag_1...g_n$ witness that a walk $a..g_b$ exists in CT. Because of this walk, we conclude that $a \prec b$ holds in CT.

$\Leftarrow$ We assume that $a \prec b$ holds in CT. Thus, an ordered-and-gate g_b with edge (b, g_b) and a walk $a...g_b$ exists in CT. We now construct a walk $a...b$ in G. In CT, an edge (a, g) is either an edge from a basic event to a gate or from a gate to another gate. Hence, a is the only basic event in the walk $a...g_b$ and we know that the other vertices g_0, ..., g_b are gates, and g_0 can exists or not. Every gate g_i in $g_0...g_b$ is an and-gate or an ordered-and-gate. By construction (line 25 and 17), every ordered-and-gate g_i is created for an action a_i in G and has an edge (g_i, a_i) in CT. An and-gate g_i is created (line 24) when a_i has several preconditions and depicts independence. We can remove every and-gate, remove g_0 when it exists and substitute every other ordered-and-gate g_i with its action a_i in $ag_0...g_b$, and result in a walk $aa_1...b$. Thus, we found a walk $a...b$ in G that ensures $a \prec b$ in G.

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

Algorithm 6.2 executes only finite loops over the actions in G in the function *createCausalTree* and creates at most once a dependency tree for every action in G. Since the actions in G are finite, Algorithm 6.2 will terminate.

Theorem 9 and Theorem 10 show that according to our correctness criteria, Algorithm 6.1 computes a DDAG G with the dependencies contained in an action trace class C. Theorem 11 shows that Algorithm 6.2 is correct and computes a causal tree CT for G that depicts the action dependencies in G. In summary, a causal tree CT computed by Algorithm 6.1 and Algorithm 6.2 correctly represents the action dependencies in C.

6.2.2 Complexity

In the following, we analyze the worst-case complexity of Algorithm 6.1 and Algorithm 6.2.

The worst-case complexity of Algorithm 6.1 is determined by the size $|C|$ of the action trace class C and the size $|A|$ of its alphabet A. Algorithm 6.1 has several computation steps of different complexity. First, Algorithm 6.1 iterates over every action trace in C and every action in an action trace (line 3-5), which has a complexity in $O(|A|{\cdot}|C|)$. In the next computation step in line 8, a lookup is executed for every tuple of two actions in $A \times A$ to create the dependency matrix `m`. Thus, the complexity to create `m` is in $O(|A|^2)$. The worst-case-complexity to compute the transitive closure is in $O(|A|^3)$ [118]. In order to ensure antisymmetricity in `m` in line 12, the algorithm compares whether the order constraints $a_1 \prec a_2$ and $a_2 \prec a_1$ of any pair of a_1, a_2 in A are in `m` and removes them in case both constraints hold. The complexity to ensure antisymmetricity is in $O(|A|^2)$. For every action in A, irreflexivity is ensured in line 13 and this has a complexity in $O(|A|)$. Next, the

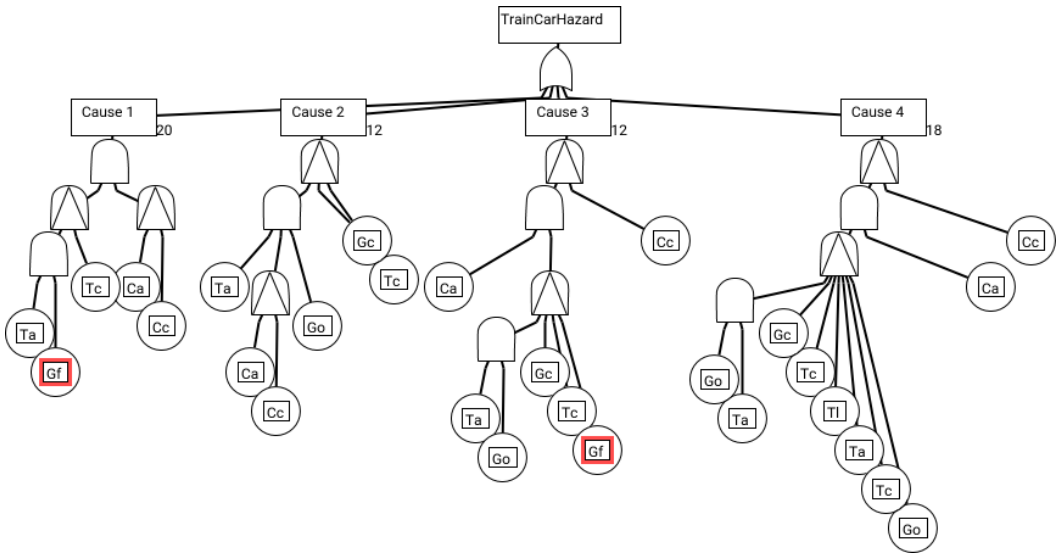


Figure 6.2: Fault Tree of Railroad Crossing with Action Order

transitive dependencies are removed in line 14. Therefore, the actions are ordered by the number of their preconditions, which has a complexity in $O(|A|^2)$ to count the number of preconditions, and a lookup happens for every triple of three different actions in $A \times A \times A$ which results in a complexity of $O(|A|^3)$. In summary, the most complex computation steps are in $O(|A|^3 + |A| \cdot |C|)$ and this is the worst-case complexity of Algorithm 6.1.

Algorithm 6.2 first determines the independent actions in G in $O(|A|^2)$. Afterwards, function `CreateTree` is called for every action a in G to depict the dependencies of a. Notice that a depends on at most $|A| - 1$ other actions. For every a, `CreateTree` creates at most one and-gate (line 24), one ordered-and-gate (line 25), and $|A| + 1$ edges. One edge starts in every action on which a depends and one edge starts in every gate that is created. This computation to depict the dependencies of a is executed at most once since the result is stored in the map `m`. We see, `CreateTree` is called $|A|$ times where every call is in $O(|A|))$ which results in an overall worst-case complexity in $O(|A|^2)$. In summary, the worst-case-complexity of Algorithm 6.2 is in $O(|A|^2)$.

6.3 Case Study

We implemented Algorithm 6.1 and Algorithm 6.2 in the tool QuantUM. We qualitatively evaluate the algorithms in that we assess whether they can jointly analyze the strict partial order in a given set of action traces. In the quantitative assessment, we measure the computing resources needed by the algorithms when analyzing a set of models. All experiments were performed on a computer with an i7-6700K CPU (4.00GHz), 60GB of RAM and a Linux operation system.

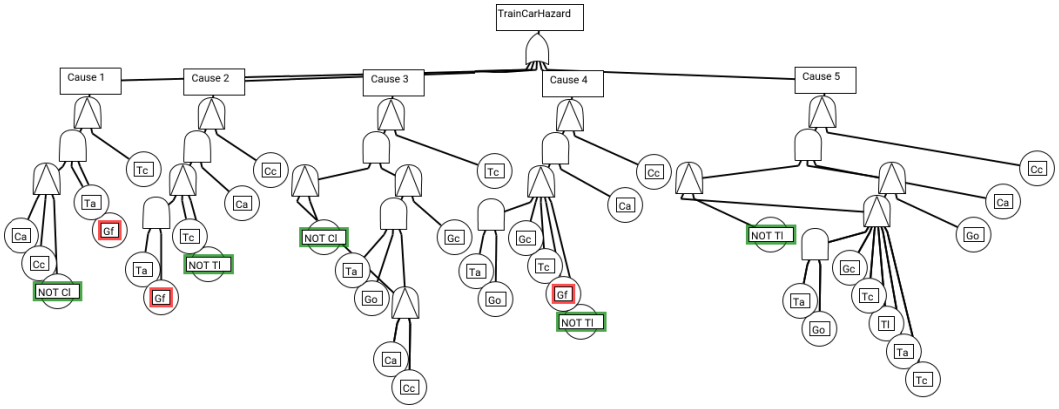


Figure 6.3: Fault Tree of Railroad Crossing with Action Order and Non-Occurrence

Qualitative Results and Interpretation. We applied QuantUM to the model of the unguarded railroad example and it computed the fault tree in Figure 6.1(c).

In Chapter 5, we analyzed Example 1 which is the railroad crossing with a gate. Without the use of the algorithms proposed in the current chapter, QuantUM computes the fault tree in Figure 5.3 that does not depict the order of the actions. Using the proposed algorithms, QuantUM computes the fault tree in Figure 6.2. It depicts the order of the actions in a cause as we defined it above. Both fault trees contain the causes `Cause 1` to `Cause 4`. In both fault trees, all actions of `Cause 1` are contained in `Cause 3` and all actions of `Cause 2` are contained in `Cause 4`. It is not clear from the fault tree in Figure 5.3 why the causal traces described by `Cause 3` and `Cause 4` are minimal according to CC1 in Definition 22. In the fault tree in Figure 6.2, we see that in `Cause 1` and `Cause 3` the gate has a failure caused by the event Gf. In `Cause 1`, the gate is stuck open and in `Cause 3` the gate first closes and then opens in error. Thus, both times the train and the car can be in the crossing at the same time, and therefore an accident can occur. In the fault tree in Figure 6.2, we see the difference between `Cause 1` and `Cause 3` in the order of the actions Gf and Tc that we assumed in Subsection 5.4.2. In `Cause 2` and `Cause 4` the system behaves without a failure of the system, but the order entailed by the interleaving of the actions causes the hazard. In `Cause 2`, the car crosses (Cc) the railroad and meanwhile, the gate closes (Gc) and the train enters the crossing. In `Cause 4`, the gate closes (Gc) and a train enters the crossing subsequently, but the signal Go to open the gate is late. Thus, the gate opens when the train is in the crossing a second time. The car can then enter (Cc) which leads to the accident. An essential difference between `Cause 2` and `Cause 4` is the order of Cc and Gc. We conclude that the ordering of the actions helps to understand causes for the occurrence of the hazards.

In Figure 6.3, we see that the proposed algorithms can also compute the order of causes according to Definition 28 that contain non-occurrence events. As proposed in Section 4.4, the analysis of non-occurrence split Cause 1 in Figure 6.2 into a Cause 1 and a Cause 2 in Figure 6.3. The Cause 1 depicts Ψ_r^1 of Formula 4.2, and Cause 2 depicts Ψ_r^2 of Formula 4.2.

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

Table 6.1: Quantitative Experimental Results of Partial Order Analyses.

Quantitative Results and Interpretation We now want to analyze the performance of the causal tree computation by Algorithm 6.1 and Algorithm 6.2. Therefore, we applied the algorithms to several models of different size, in terms of the number of states and transitions that they encompass, taken from [102] and Chapter 7. The quantitative results are given in Table 6.1. The complexity of a model is given in terms of the number of its states and transitions. For every model, we indicate the number $\#C$ of causes and the maximal number of causal traces and actions in one of the causes. The columns Time and Memory indicate the maximal computation time and memory consumption that the analysis required in order to compute a fault tree of a model including the action order as per the proposed algorithms.

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

The diagram in Figure 6.4 gives the time in microseconds (μs) that is necessary to compute a causal tree for every cause in every model. For a model with several causes, the diagram depicts several data-points. The worst-case complexity to compute a cause is the combined worst-case complexity of Algorithm 6.1 and Algorithm 6.2 and is in $O(|A|^3)$. We let IBM SPSS [72] analyze the cubic relation between the time to compute a cause and the number of actions. IBM SPSS automatically fits the function $31.583 + 13.583x - 1.057x^2 + 0.057x^3$ to the data. It is depicted as a black line in the diagram. The distance of the points to the function can be measured by the coefficient of determination R^2, which is the quadrate of the correlation. The value range of R^2 is $[0, 1]$ where $R^2 = 1$ would be a perfect fit. The function in the diagram has a $R^2 = 0.985$. This function fits nearly perfectly to the data points, which supports that the worst and average case complexity of the proposed order analysis has a cubic complexity.

While the time to compute a causal tree is given in Figure 6.4 in microseconds, the overall time to compute a fault tree is in Table 6.1 in the area of milliseconds. We wondered about this gap of a factor 100 and detected that Java, which was used for the implementation of QuantUM and the proposed algorithms, has an offset time in the area of milliseconds to load and create a class when the class is instantiated the first time. This implies that the overall computation times given in Table 6.1 consist primarily of the time for loading classes and not of the time for computing the causal trees.

Our proposed algorithms computed the strict partial orders within at most 40 milliseconds and at most 17.038MB of memory. This seems reasonable and

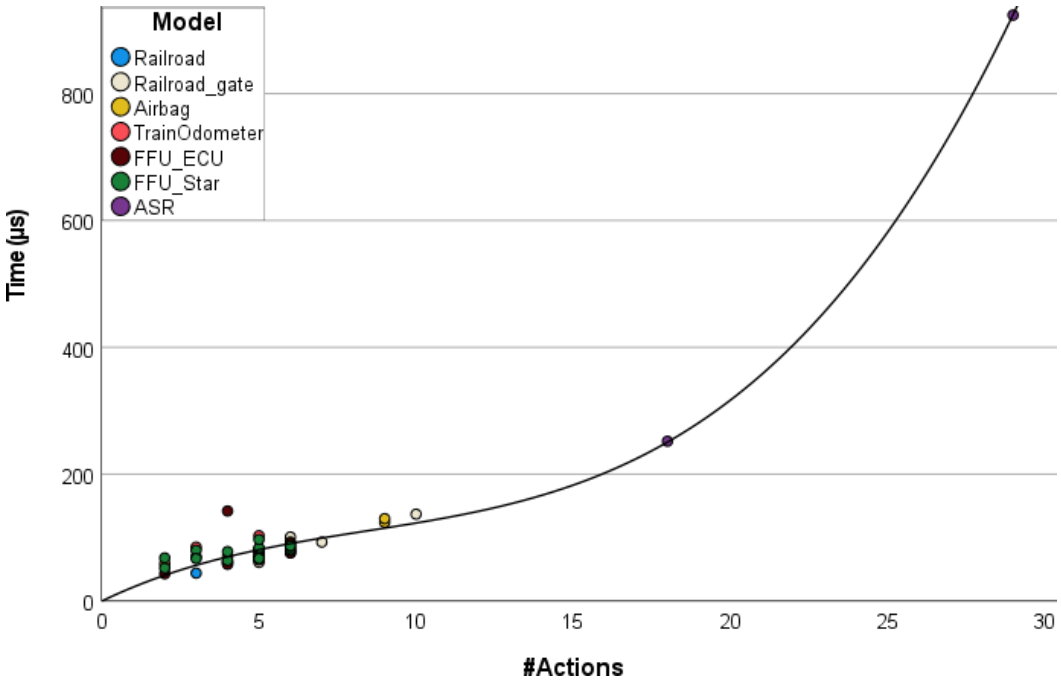


Figure 6.4: Time to Compute a Causal Tree in Relation to #Actions in a Cause.

is acceptable for QuantUM since a cause in the analyzed models contains up to 61,920 traces of 29 actions.

6.4 Conclusion

In this chapter, we presented algorithms that compute a strict partial order of the actions occurring in an action trace class and represent this strict partial order as a fault tree. In Causality Checking, we use the algorithms to compute strict partial orders of causal traces with the same action set. These strict partial orders represent causes. We implemented the algorithms in the tool QuantUM and computed fault trees for several models. The evaluation showed that a representation of the action order can be computed using a reasonable amount of computing resources, and that the computed results provide helpful insight into the causes for a property violation.

Chapter 7

Automated Functional Safety Analysis of Automated Driving Systems

The content of this part is based on the publication [86].

Contents

7.1 Introduction

We now show in a case study on automated driving systems (ADS) that Causality Checking implemented in the tool QuantUM supports the analysis of safety-critical systems. An ADS system steers a vehicle without a driver and it is essential for these systems to ensure functional safety, which means, that a function does not lead to a hazard even in the context of errors. Although functional safety of vehicles is ensured by the ISO 26262 [76] standard, this standard is not directly applicable to ADS. In the following, we describe the fundamental changes in the safety-analysis of an ADS, and present a safety-analysis for ADS in the "sense" of the ISO 26262 standard. The realization of the analysis shows that QuantUM supports the safety analysis according to the ISO 26262 standard and we also extend QuantUM to support analysis required for ADS.

Changes in the Safety Analysis due to Autonomous Systems. The functional safety of software-driven functions in passenger vehicles is currently the subject of the ISO 26262 [76] international standard. It specifies development processes and requirements ensuring functional safety of software defined safety-critical functions, also referred to as items, in automobiles. The ISO 26262 standard focuses primarily on the safety of the software-defined items in the presence of systematic software and random hardware faults.

The advent of assisted and autonomous driving is fundamentally changing the architecture of software-defined critical automotive systems. As a consequence the methodological foundations of asserting functional safety of such systems will have to be redeveloped. The current version of the ISO 26262 standard, as well as the current proposed revision on this standard [77], do not account for the functional safety of autonomous driving functions.

First, the development of autonomous driving systems (ADS) will at some point lead to vehicles in which a human driver will no longer be available to take over control of the vehicle. Following the classification in the SAE J3016 standard [123], this will be the case starting at level 4. Whereas classical functional safety approaches follow a *fail-safe* approach, which in case of a failure relies on a driver being able to take over control of the vehicle and bring it into a safe state, ADS systems have to be designed to operate in a *fail-operational* manner. This means that in the presence of the failure of some ADS function, the overall vehicle system will remain operational for a certain period of time, with a given probability, in order to navigate the vehicle automatically into a safe location, for instance, the shoulder [139]. This is frequently also referred to as "limp-home" mode. For the analysis of functional safety properties this means that the availability of these limp-home mode functions in the presence of a system failure needs to be proven.

Second, the conventional approach to functional safety, as reflected by the current ISO 26262 standard, is highly "item-oriented". This means in particular that one driving function, or item, is implemented by one software component executing exclusively on one hardware unit, referred to as electronic control unit (ECU). Current systems already break with this strict concept and run a low number of functions on a single ECU. However, safety arguments largely rely on execution in isolation, with the exception that some degree of freedom of interference, including that caused by concurrency problems, at the level of the underlying execution platform has to be proven. This will not be the appropriate paradigm for ADS. In those systems, many sub-functions will co-operate and be highly interdependent in order to implement an overall system function, namely to drive safely from location A to location B [111]. Furthermore, for cost, performance, flexibility and dependability reasons, ADS will be implemented on networked computing platforms that encompass a low number of processors, connected by high bandwidth real-time networks, and potentially possessing multiple cores [101]. To increase reliability, redundant software functions can be mapped to different hardware components, both statically and possibly also dynamically. As a consequence, many functions will be mapped to a single or more hardware components, which means that a software-hardware mapping problem needs to be considered in the system and safety design. Again, current ISO 26262-type functional safety analyses do not account for this type of architectures.

Third, ADS will be highly concurrent, due to the parallel processing of sensor data and decision making to support different driving functions, leading to concurrency non-determinism. Another change with ADS is the application of non-linear machine-learning algorithms based on neural networks that are heavily used in environment perception. Non-determinism and non-linearity make it particularly difficult to use classical safety analysis techniques, such as Fault Tree Analysis (FTA)

or Failure Mode and Effects Analysis (FMEA) proposed for piloted driving in ISO 26262, in a non-automated, manual fashion.

Contributions of the Chapter. In this chapter, we propose a method to analyze functional safety of ADS "in the spirit" of ISO 26262, to the extent that it is applicable, and address some of the challenges pointed out above. The method is model-based and relies on SysML [115] models that describe the nominal and the failure behavior of components, as well as software-hardware mappings. We embed these models into the QuantUM method and tool for analyzing causes of safety violations. QuantUM employs Causality Checking and the probability computation presented in [105] in order to compute ordered sequences of events that are deemed to be causes for safety violations. We depict these causes in a probabilistic fault tree. The benefits of this approach include the following aspects.

- The algorithmic model analysis methods employed in QuantUM (model checking, Causality Checking) are well suited to deal with concurrency induced non-determinism. Dealing adequately with the non-linearity caused by using neural networks based machine learning is not addressed in this chapter.

- The proposed analysis avails itself to an implementation in an automated software tool. Once the models and properties are defined, the analysis performed by QuantUM requires no further interaction with an engineer.

- The SysML models can easily be modified, for instance to analyze architectural alternatives as well as alternative software-hardware mappings during design space exploration. The functional safety analysis can then easily be repeated at little cost by invoking QuantUM on the modified model.

- The developed tools can be qualified according to, for instance, ISO 26262.

We evaluate our approach by applying it to a case study in which we perform a functional safety analysis for a practical ADS architecture [23], for which we analyze two mappings of ADS functions to hardware. The analyzed system failures can be used to assess the impact of single or multiple faults on the overall failure probability, as requested by ISO 26262. The analysis also enables an engineer to select efficient failure handling concepts and to evaluate different possible architectures while meeting safety goals as specified by ISO 26262.

Structure of the Chapter. In Section 7.2, we present the demands of ISO 26262 on vehicles and the change in the architecture with the development of ADS. In Section 7.3, we explain the analysis steps to verify an ADS architecture in the "spirit" of ISO 26262 by QuantUM, and a necessary extension of QuantUM. In Section 7.4, we illustrate our approach by applying the steps on two ADS architectures. In Section 7.5, we draw conclusions.

7.2 ISO Standard 26262

Functional Safety and Autonomous Driving. The ISO standard 26262 [76] as well as its recently proposed revision [77] define requirements on software development processes for safety-critical functions of an automotive passenger vehicle. This is to ensure that the functional safety of a passenger vehicle is challenged by no more than an acceptable residual risk. The standard is focused on mechanisms that ensure functional safety of critical software-driven functions in the presence of systematic faults and random hardware faults. It assumes that systematic faults can be eliminated by verification and validation techniques, in particular testing. The standard does not tackle random software failures, which happen non-deterministic, for instance, due to concurrency issues or special environment influences. Notice that the ISO 26262 standard does not address techniques to ensure "Safety of the Intended Function" (SOTIF) [78], i.e., the safety of intended functionalities of the vehicle itself.

Two characteristics of the ISO 26262 standard are important in the context of this work.

1. The standard is "item-oriented", which means that it addresses safety mechanisms for items, such as airbag control, steering, braking, light control, etc., in isolation. This approach is inappropriate for ADS since different vehicle functions will be interdependent by acting as backup functions for others. Further, the driver as the function integrator and coordinator in piloted driving is not available, which means that the software has to take over these integration functions.

2. Neither the published version of the ISO 26262 standard nor the its proposed revision address assisted or autonomous driving per se [94]. To the contrary, the ISO 26262 standard allows safety mechanisms to rely on the driver taking over control of the vehicle in order to mitigate the impact of function failures, which in ADS at SAE level 3-5 is likely not to be practical [137].

Nonetheless, we show how formal analysis techniques can be used to support the safety engineering of ADS in the spirit of ISO 26262. According to ISO 26262, different driving functions, referred to as "items", are assigned an Automotive Safety Integrity Level (ASIL), ranging from the least critical ASIL A to the most critical ASIL D. The determined ASIL implies the design methods that are to be used. As argued above, we will consider the ADS driving function as a unique "item" in the ISO 26262 sense and perform a safety analysis on this set of functions as a whole, including an assignment of an ASIL.

According to ISO 26262, safety goals need to be defined to ensure that the failure probability of software functions in the presence of random hardware faults lies at an acceptable minimum. For each ASIL, a maximum tolerable probability of failing a safety goal due to random hardware faults is specified. A system failure may be a result of a single fault (single-point failure) or a combination of faults (multiple-point failure). From the safety goals, functional safety requirements are derived. In safety analyses one will also have to consider fault rates of the underlying

hardware, for instance sensor faults, as well as the hardware-specific fault detection rates. Those will later occur as parameters of our models.

Following ISO 26262, the system architecture design is derived from technical safety requirements. For ASILs A to D, the standard recommends documenting the architecture using a semi-formal notation, such as the SysML, for ASIL A and B, and strongly recommends this for ASIL C and D [77, Part 6]. The system architecture then needs to be verified against the safety requirements [76, Part 4]. The process mandated by ISO 26262 for this verification includes a system design analysis to identify possible effects of faults, the causes of possible failures and a quantification of failures. Applicable methods include Fault Tree Analysis (FTA) and Markov models. The use of formal verification techniques, including model checking [14], is recommended for software architecture verification for ASIL C and D [76, Part 6]. This includes verification that certain safety goals are met by a given system design [76, Part 8]. We propose that an automated approach based on a formal analysis of the state space described by the system architecture helps to detect and explain safety goal violations at an early stage in the safety engineering process, thereby meeting the requirements of ISO 26262. It also enables automated, tool-supported architectural variant analysis during safety and system engineering, greatly contributing to reducing the related costs.

System Architectures for ADS. A functional architecture for autonomous driving is proposed in [23]. The authors extract, from several conceptual as well as practical implemented architectures, a layered architecture. The semantics understanding of the external world is calculated in the *perception layer*. It computes an external world model based on a fusion of the various forms of external sensor information that it receives. The external world model in conjunction with the internal state of the car, which is defined among others by the energy management and failure states of the platform, are used by the *decision and control layer* to make decisions about the execution of a trajectory. The trajectory is then used by the *vehicle platform manipulation* layer to drive the actors, like steering and braking, and keeping the platform overall stable. All three layers have a complex structure of interdependent, cooperating elements, each representing a specific function.

Functional Safety Goals for ADS. A predominant idea in ISO 26262 is that a system needs to reach a safe state in the event of a system failure, in other words, that it is *fail-safe*. When the driving is piloted, this can often be achieved by switching the defective subsystem off and leaving it up to the driver to deal with the situation. In autonomous driving, this option does not exist, as argued above. The objective here needs to be that in the presence of the failure of one function in an ADS, the overall system architecture needs to remain operational for a certain period of time to ensure that a safe state can be reached. This capability is often referred to as "fail-operational". The ISO 26262 standard states: "If a safe state cannot be reached by a transition within an acceptable time interval, an emergency operation shall be specified." [76, Part 3]. This means that designing safety mechanisms that ensure a limited backup capability for a defective functionality for a certain period of time is within the practices recommended by ISO 26262. A typical example would be that

the braking system takes over functionalities of a failed steering control system by applying differential torque or braking for a limited period of time so as to "limp home" to a safe part of the road, such as the shoulder. The safety goals that we pursue in our analysis will, hence, have to reflect the probabilities of remaining *fail-operational* for a certain period of time.

7.3 Safety Analysis of an ADS Architecture

Safety Goals for an ADS. Following the earlier made argument, we consider the driving function of an ADS to be one item, i.e., one driving function. Using this assumption we perform a safety analysis for this item in the spirit of ISO 26262.

As argued above, we need to consider a fail-operational architecture. When reaching a failure state, the ADS reacts by switching to an emergency mode that handles the failure situation. For a safety analysis of an ADS, we consider possible hazards of an architecture and derive appropriate safety goals to prevent the hazards:

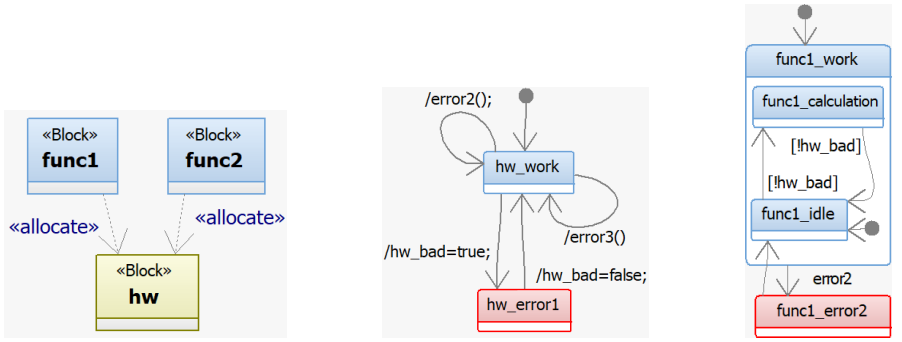
1. When a vehicle is operating as an ADS it has to control the vehicle platform even if it is in an emergency mode. If the control is lost, the vehicle will crash. To prevent this hazard we derive the safety goal SG1: *Ensure that the ADS provides driving information to the vehicle platform at any time.*

2. The ADS can have an undetected failure. As a consequence, the emergency mode may not be activated. The detection of a failure ensured by the safety goal SG2: *Ensure that the emergency mode is enabled when a failure of the ADS occurs.*

3. If the system cannot enter or remain in the emergency operation mode for a specified period of time, a safe state may not be reachable. We assume the period of time necessary to reach a safety state to be t_1 seconds and derive the safety goal SG3: *Ensure that the emergency mode of the ADS is available on demand for at least t_1 seconds.*

The ASIL classification of a safety goal is determined according to the severity of a function failure caused by a hazard, the probability of exposure to a situation with a potential failure, and the controllability of the failure situation by the driver. We assume the severity of each hazard of the ADS to be potentially life-threatening (S3 according to [76, Part 3]). Since the ADS system will be active most of the time when the vehicle is in operation, certainly during more than 10% of the operation time, we assume the probability of exposure to be high (E4 according to [76, Part 3]). We also assume the controllability to be very low (C3 according to [76, Part 3]), since in the case of SAE level 3 driving the driver may be surprised by a failure situation, or unable to handle it due to the low occurrence rate of such a situation. These valuations hold for all three safety goals and consequently this implies, according to ISO 26262, an ASIL-D classification for each safety goal.

As argued above, ISO 26262 recommends the use of formal methods, including model checking, for the analysis of ASIL D safety goals.



(a) Mapping 1 (b) STM `hw` (c) STM `func1`

Figure 7.1: ADS Example Modeling

Automated Safety Analysis of an ADS Architecture. We now describe how an extended version of the QuantUM tool can be used to perform an automated safety analysis for a given ADS architecture with respect to safety goals SG1 - SG3.

Step 1: ADS Modeling. The system architecture of a vehicle consists of several software function units executing on a number of hardware units. Each unit is represented by a block in a SysML BDD, see the example in Figure 7.1(a). It depicts two software function units, represented by blocks colored blue, executing on one hardware unit, colored yellow. Assigning a software function to a hardware unit on which the function is executed is referred to as software-hardware mapping. In the BDDs, mappings are depicted using dashed arrows with the label ≪`allocate`≫. The behavior of each unit is modeled by a SysML STM diagram. The STMs of different units execute concurrently. For the example in Figure 7.1(a) the behavior of the blocks `hw` and `func1` are exemplified in Figures 7.1(b) and 7.1(c), respectively.

In the STMs, the blue states represent the nominal behavior of the units, and the red states represent their failure behavior. The state machines execute their normal behavior by staying in a "work" state. To reach a failure state, a fault event has to occur. The first type of fault directly leads to a failure of the unit. This can, for instance, be caused by a loss of power, or by a permanent error such as a broken hardware element. These faults are modeled by a transition to a failure state, such as `hw_error1` in Figure 7.1(b). The unit remains in this state until it is repaired, represented by a repair transition to the work state. As a result of entering a failure state, a hardware unit stops executing and any software function, executing on this hardware unit, will cease to operate as well. To model this behavior, the Boolean variable `hw_bad` is set to true and all transitions of the function unit are disabled by a guard `!hw_bad` (see Figure 7.1(c)). The second type of fault leads to an error inside of the hardware unit, such as a bit flip, even though the unit continues to operate. The hardware unit is not corrupted, but an error is propagated to functions executing on that unit. In the SysML model, error propagation is modeled by message passing. In the example in Figure 7.1(b), two errors are propagated by messages `error2` and `error3` to the respective software functions. With the receive

of such an error function 1 enters the `func1_error2` state and function 2 enters the `func2_error3` state upon receiving `error3`. From these states, the function can return to its normal behavior by a transition representing failure repair.

Step 2: System Failure Modeling. The ADS fails if one of the safety goals SG1-SG3 is violated. The violation of these system goals needs to be mapped to states that the different STMs in the system are entering. QuantUM offers the possibility to tag states in the STMs of different blocks as error states, and then permits to either use a logical *or* or a logical *and* between all tagged states in order to characterize a violation state of the system. To model the safety goals needed, we extend this rather inflexible scheme. An ADS has the structure of a set of channels. Sensory input data is processed by a function and the output data is forwarded to the next function, forming one channel called the primary channel. The emergency mode adds a second, partly redundant backup channel to the ADS. The ADS fails and violates its safety goals if there is a function failure in each of the channels. We attach a Boolean variable `bad` to each function and permit forming logical expressions on these variables to express the failure of one channel. We combine the failure expressions of each channel with an "and" and add the result to the property. For example, in order to check two redundant channels the resulting property has the form *it is never the case that step1 or step2 or ... of channel1 is bad and step1 or step2 or ... of channel2 is bad.*

Step 3: Analysis of Emergency Mode Failures. A violation of SG3 implies that the normal ADS behavior has a failure and either the emergency mode functionality is not available on demand, or it is not provided for at least a certain period of time and therefore constitutes a fundamental challenge to the safety of the vehicle. In the following we compute the probability P_{fail} for a violation of SG3. The analysis performed by QuantUM works on a global state graph obtained by interleaving the local behaviors of the concurrent system hardware and software components. A path in this graph, representing an execution of the ADS, constitutes a violation of SG3 if in a state the emergency mode is being activated but not going to be available for at least a period of time t_1. We characterize the set of all emergency activation states S in the global state graph using a Boolean expression e formed as described in Step 2. In accordance with the foundations of probabilistic model checking we will consider reaching a first state $s_i \in S$ as a stochastic event, with the path consisting of a stochastic experiment. The event of reaching a state s_i first, denoted by $reach\_s_i$, precludes the event of reaching another state $s_i' \in S$ first, which means that stochastic events we consider do not overlap. As a consequence, we may partition the sample space, which consists of all possible paths in the global state graph, according to the events $reach\_s_i$. In a first probabilistic model checking step performed by QuantUM, we compute the probability $P(reach\_s_i)$ to reach each state in S within a period of a driving cycle t_dc. In a second model checking step we compute the probability $P(fail_i|reach\_s_i)$ to reach a failure from state s_i within a time t_1. To enable the first model checking step we change the model in such a way that we conjoin $\neg e$ with all transition guards. This means that when the system enters a state in which e becomes true, this state is turned into an end state with no

enabled exit transition. For each end state we calculate its probability $P(reach\_s_i)$. For the second model checking step, we compute the probability $P(fail_i|reach\_s_i)$ by starting in any state defined by expression e. The probability P_{fail} is computed by a summation over all products of $P(fail_i|reach\_s_i) \cdot P(reach\_s_i)$, which is justified by the memoryless nature of CTMCs. No Causality Checking will be performed and no fault trees will be computed by QuantUM during SG3 violation analysis.

Step 4: Probability Rates. Probability rates, in particular for hardware failures, repairs and failure detection, are difficult to determine and usually depend on a specific domain and the concrete hardware used. However, even if precise rates are not available, the comparison of the relative failure probabilities of architectural variants with identical and with different estimated or assumed rates can be of great importance. This can, for instance, answer the question of how architectural variants will affect failure probabilities, or what error detection rates are required to achieve the desired level of failure probability. In QuantUM, the SysML model is labeled with probability rates, for instance, for the probability of executing a failure or repair transition. QuantUM uses probabilistic model checking, in particular model checking for Continuous Time Markov Chains (CTMCs) [13], in order to compute the probabilities for the causes leading to a violation of safety properties. A fault event of the hardware may lead to different faults in a system. In this situation, we distribute the fault rate over the different fault transitions. The portion of the fault rate that each transition receives relies on domain specific knowledge that the designer needs to provide. For example, in Figure 7.1(b), a bit flip with a probability rate of 10^{-4} can cause an `error2` or an `error3`. Notice that throughout the chapter, rates are assumed to be per hour. Assume that it is typical that 40% of the errors are of type `error2` and 60% are of type `error3`. This leads to a fault rate of $0.4 \cdot 10^{-4}$ for `error2` and a fault rate of $0.6 \cdot 10^{-4}$ for `error3`. To split the fault rate in this way is appropriate for CTMCs, cf. [13].

A potential threat to the validity of the failure probabilities computed by QuantUM and the probabilistic model checker Prism [97] that QuantUM uses, is the fact that the original SysML model mixes non-probabilistic and probabilistic transitions. For the non-probabilistic transitions Prism assumes a default rate of 1. Assuming that we consider one time step, based on the negative exponential distribution on which CTMCs are based this translates into a probability of less than 1 of taking this transition with which the accumulated path probability up to this step will be multiplied. However, we do not experience a negative effect on the total failure probability since the SysML model structure that we propose implies that the system will cycle through non-probabilistic normal behavior, for which the path probability is 1, until it performs one probabilistic failure transition to enter a failure state. For example, the state `func1_work` in Figure 7.1(c) has non-probabilistic transitions between the states `func1_calculation` and `func1_idle` with a default rate of 1, remaining in state `func1_work` until the probabilistic transition `error2` is taken.

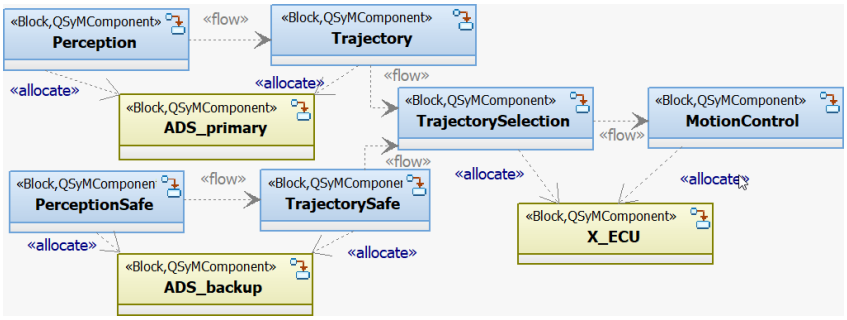
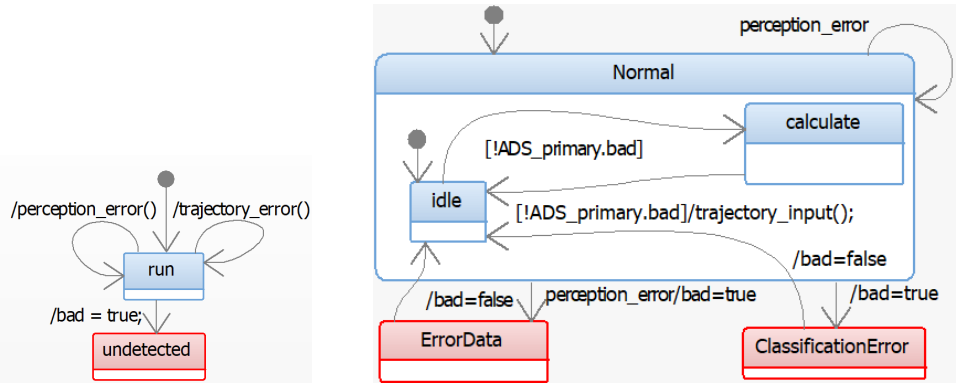


Figure 7.2: Architectural Variant 1 for ADS

7.4 Case Study: A Comparison of Autonomous Driving Architectures

Step 1: ADS Modeling. [23] proposes a functional architecture generalized from real architectures. We use part of this functional model and add several hardware units. The resulting mapping problem leads to a number of architectures. The SysML BDD in Figure 7.2 gives an overview of the structure of the first architectural variant that we consider. We model the perception layer by a block `Perception` and the motion and control layer by a block `Trajectory`. Since the functions represented by these two blocks are critical for the proper functioning of the ADS, we add blocks `PerceptionSafe` and `TrajectorySafe` to provide redundant backup functionality. The function represented by the block `Trajectory_Selection` selects by default the trajectory of block `Trajectory`, but switches in case of a failure of these blocks to the alternative trajectory computed by block `TrajectorySafe`. The block `MotionControl` represents the interface with the vehicle platform manipulation layer by providing it with control parameters, such as steering angle, braking force or differential torque, that the vehicle platform will translate into commands for the actuators of the vehicle. Figure 7.2 also illustrates the software-hardware mapping that we propose for the first architectural variant. Notice that the primary functionalities for perception and trajectory computation are mapped to the hardware block `ADS_primary`, while the backup functionality `ADS_backup` is mapped to a separate hardware unit `ADS_backup`, thus increasing the probability that the backup functionality will be available even in the case of a failure of the primary hardware represented by block `ADS_primary`.

The state machine modeling the behavior or block `ADS_primary` is given in Figure 7.3(a). The hardware operates correctly in state `run`. In this state, the occurrence of a detected error inside the hardware, for instance a memory bit flip, is communicated to the `Perception` block using a `perception_error` message in case the perception function is currently executing on `ADS_primary`. In case a trajectory computation function is executing, the hardware error will be communicated using a `trajectory_error` message to the `Trajectory` block. If in the `run` state an undetected hardware error occurs, the impact on the hardware is unknown. We model this by a transition into state `undetected` along which we set the failure



(a) STM `ADS_primary` (b) STM `Perception`

Figure 7.3: ADS STM Examples

variable `bad` to true. In this state, no software function can be executed on the hardware.

The behavior of the block `Perception` is defined by the hierarchical state machine in the STM diagram in Figure 7.3(b). The normal behavior is modeled in the nested state `Normal`. The function starts its computation in the state `idle` and cycles through states `idle` and `calculate`, which represents the processing of sensor information, as long as the variable `ADS_primary.bad` is false. When returning to idle it sends the message `trajectory_input` to the function trajectory in order to indicate that input data for the trajectory function is available. Upon receipt of a message `perception_error`, the perception function can decide either to handle this message and remain in the `Normal` state, or it can decide to enter the `ErrorData` state and set its bad value to true. Upon repair it can return to the `idle` state and resume execution. Deviating from the classical ISO 26262 viewpoint to only consider hardware failures, we also consider software failures. Such a failure in the block `Perception` is modeled by a non-deterministic group transition from the `Normal` state to the `ClassificationError` state in the course of which the `bad` variable will be set to true. In this way, we can model, for instance, classification errors in neural networks that are commonly used for perception. We assume that these errors can also be repaired, modeled by a return to the `idle` state. The other blocks representing hardware and software functions have behaviors similar to the ones described above.

Step 2: System Failure Modeling. We characterize a system failure of the ADS violating SG1 or SG2 using different Boolean expressions for each architectural variant and safety goal. The Boolean propositions refer to the `bad` variables of the blocks in the SysML model. For architectural variant 1, a system failure violating SG1 happens when a software function or a hardware unit fails in both channels, or when at least one of the functions `TrajectorySelection` or `MotionControl` or

the hardware `X_ECU` fails. This leads to the Boolean failure expression

$$\begin{aligned}&((\mathit{ADS\_primary.bad} \vee \mathit{Perception.bad} \vee \mathit{Trajectory.bad})\\ &\wedge(\mathit{ADS\_backup.bad} \vee \mathit{PerceptionSafe.bad} \vee \mathit{TrajectorySafe.bad}))\\ &\vee \mathit{TrajectorySelection.bad} \vee \mathit{MotionControl.bad} \vee \mathit{X\_ECU.bad}.\end{aligned} \tag{7.1}$$

Architectural variant 2 differs from the first in that the block `TrajectorySelection` is mapped to the block `ADS_backup`. As a consequence, architectural variant 2 fails under the same failure condition as variant 1, and additionally by a failure of `ADS_backup`. This leads to the Boolean failure expression

$$\begin{aligned}&((\mathit{ADS\_primary.bad} \vee \mathit{Perception.bad} \vee \mathit{Trajectory.bad})\\ &\wedge(\mathit{ADS\_backup.bad} \vee \mathit{PerceptionSafe.bad} \vee \mathit{TrajectorySafe.bad}))\\ &\vee \mathit{ADS\_backup.bad} \vee \mathit{TrajectorySelection.bad}\\ &\vee \mathit{MotionControl.bad} \vee \mathit{X\_ECU.bad}.\end{aligned} \tag{7.2}$$

The function `TrajectorySelection` is responsible for selecting the emergency trajectory in case of a failure. The function fails if the function itself or the underlying hardware is in a failure state. This leads to a violation of SG2, expressed by the Boolean failure expression $\mathit{TrajectorySelection.bad} \vee \mathit{X\_ECU.bad}$ for architectural variant 1 and $\mathit{TrajectorySelection.bad} \vee \mathit{ADS\_backup.bad}$ for variant 2.

Step 3: Analysis of Emergency Mode Failures. For the computation of the failure probability of the emergency mode we need to determine the expected operation time of the emergency mode t_1, a Boolean expression representing a failure of the ADS and a Boolean expression characterizing the activation of the emergency mode. We assume t_1 to be 10 seconds. The failure states of the ADS for the two architectural variants are encoded by the Boolean expressions 7.1 or 7.2, respectively. The emergency mode of the ADS is activated in both architectural variants if a software function running on hardware `ADS_primary` or the hardware `ADS_primary` itself fails. We encode these states using the Boolean expression $(\mathit{ADS\_primary.bad} \vee \mathit{Perception.bad} \vee \mathit{Trajectory.bad})$.

Step 4: Probability Rates. In order to determine rates in the context of our case study, we assume `ADS_primary` and `ADS_backup` to be implemented using "standard" hardware without hardware checks in order to meet the high computing power demands of the software functions executing on them. In such hardware components most faults happen because of memory errors [127], and we assume typical fault rates of 10^{-4}. Since there are no special computing power demands that apply to `X_ECU` and since we have not accounted for any redundancy here we assume safety hardware to be used with a fault rate of 10^{-8}. We further assume a hardware fault detection rate of 99%, i.e., 1% of the errors remain undetected. As described above, the probability of detected hardware faults are distributed evenly over all software functions running on the considered hardware unit. For instance, `perception_error` and `trajectory_error` are assumed to each have a probability

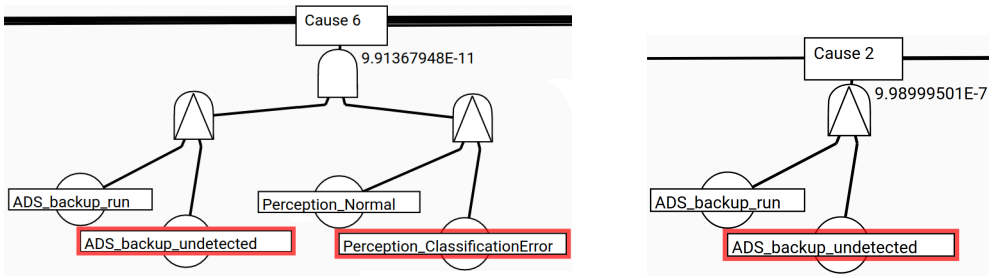
	Architectural Variant 1			Architectural Variant 2		
Safety Goal	Memory	Time	States	Memory	Time	States
SG1	624.03MB	33:57:51	226,688	588.01MB	25:23:56	199,206
SG2	570.10MB	3:37:13	311,232	574.16MB	5:59:47	315,150
SG3	64.67MB	5:58	170,182	104,00MB	4:46	129,441

Table 7.1: Computational Effort for SG Violation Analyses

of 49.5%, i.e., a rate of $0.495 \cdot 10^{-4}$. We assume that a software function affected by a detected hardware error handles the error with a probability of 90%, but will fail with a probability of 10%. For software failures of the perception function, we assume a fault rate of 10^{-4}. Functions in a failure state can resume by a repair transition. We assume a repair rate of $4 \cdot 10^{-2}$ for software functions (cf. [54]).

Analysis using QuantUM. We assume a driving cycle duration t_{dc} of 1h in all of the analyses. The result of the analysis for violating SG1, and thereby SG2, is a fault tree with the state representing the SG1 violation as top level event. The fault tree in Figure 7.5 depicts the 19 causes for architectural variant 1, and the fault tree in Figure 7.6 depicts 16 causes for architectural variant 2. Since the fault trees are very large, we use in the discussion the excerpt in Figure 7.4. Architectural variant 1 has the probability of $1.31998187 \cdot 10^{-8}$ and architectural variant 2 the probability of $4.30677042 \cdot 10^{-6}$ to violate SG1. Due to the redundant structure of the architecture in both variants, analyzing SG2 in isolation leads to two single source failures already detected by SG1. One single source failure involves `trajectory_selection_error` for both variants and the other failure involves `X_ECU_undetected` for variant 1 and `ADAS_backup_undetected` for variant 2. The probability of a violation of SG3 is $6.399474 \cdot 10^{-9}$ for variant 1 and $6.164128 \cdot 10^{-9}$ for variant 2.

The experiments were performed on a computer with an i7-6700K CPU (4.00GHz), 60GB of RAM and a Linux operation system. The computational efforts in memory, time and for the architectural variants are depicted in Table 7.1. The column *States* gives the number of states explored by QuantUM, in the case of SG3 this only comprises the number of states analyzed by Prism.



(a) Cause 6 of Architectural Variant 1 (b) Cause 2 of Architectural Variant 2

Figure 7.4: Excerpt of ADS Fault Trees

Result Interpretation. The architectural variant 1 has a lower probability of violating SG1. This result can be explained as follows. The fault trees for the two architectural variants differ mainly in the probability of the causes that contain the event `ADS_backup_undetected`, representing an undetected hardware failure in the hardware unit that is subject to the altered software-hardware mapping. The fault tree of architectural variant 1 contains four causes that contain the event `ADS_backup_undetected`, of which one cause is depicted in Figure 7.4(a). With a probability of $9.91367948 \cdot 10^{-11}$, it does not contribute significantly to the total SG1 violation probability. All other causes that contain `ADS_backup_undetected` have a similarly insignificant probability. Notice that failure events in the fault trees are marked in red.

The fault tree of architectural variant 2 contains one cause with this event, depicted in Figure 7.4(b), with a probability of $9.98999501 \cdot 10^{-7}$, thus contributing significantly to the SG1 violation. While in architectural variant 1 the `ADS_backup_undetected` fault needs to coincide with a `Perception_ClassificationError`, in architectural variant 2 the occurrence of `ADS_backup_undetected` suffices to lead to an SG1 violation. The difference in the probabilities of the two considered causes is due to the fact that the conditional occurrence of two failure events, such as in Cause 6 of architectural variant 1, is less probable than the unconditional occurrence of a fault event as in Cause 2 of architectural variant 2. Due to the software-hardware mapping in architectural variant 2, the fault event `ADS_backup_undetected` directly leads to an SG2 violation, and this scenario has a high probability. The difference in the probabilities of SG1 violations can hence be traced back to the difference in the hardware-software mappings used in both architectural variants.

In the following, we discuss the influence of the detection and error handling rates. We first increase the detection rate of `ADS_backup` for variant 2 from 99% to 99.99%. The higher detection rate decreases the failure probability of Cause 2 from $9.98999501 \cdot 10^{-7}$ to $9.98999995 \cdot 10^{-9}$. This change has no significant effect since the probability of reaching error state `undetected` is decreased by the same amount that the error probability of the functions running on the hardware is increased. The probability of violating SG1 is now mainly due to reaching failure state `ErrorData` of function `TrajectorySelection`, which is $3.32799547 \cdot 10^{-6}$. In a second step, we increase the error handling rate of function `TrajectorySelection` from 90% to 99%. This decreases the failure probability for `ErrorData` in function `TrajectorySelection` to $3.32805910 \cdot 10^{-8}$. As a consequence the overall failure probability of violating SG1 decreases from $4.30677042 \cdot 10^{-6}$ to $5.60069041 \cdot 10^{-8}$. We notice that detection and error handling rates have an essential influence on the failure probability of the ADS.

A violation of SG3 is less probable than 10^{-9} for both variants. The small probabilities are reasonable since the ADS remains in the emergency mode for only 10 seconds, which is much shorter than the assumed driving cycle of one hour. Unexpectedly, a violation of SG3 is more probable for variant 1 ($6.399474 \cdot 10^{-9}$) than for variant 2 ($6.164128 \cdot 10^{-9}$). The difference is due to the fact that variant 2 fails with higher probability without entering the emergency mode.

The presented architecture variant do non-deterministically violate the prop-

erty SG1 because of the self-transitions in the normal behavior. By deleting these transitions, we get architecture variants that deterministically violate SG1 and also analyze non-occurrence in the cause computation. The fault tree of a deterministic variant 1 has then 28 Causes and of a deterministic variant 2 has 25 Causes. In the fault tree, a cause contains a non-occurrence event when it contains a software failure behavior. In both variants, this is reasonable, since only the red states of the software modules have repair transitions. When a cause without considering non-occurrence contains two software failure behavior, then analysis of non-occurrence results in two causes, each with a different non-occurrence event. This explains the higher number of causes in the deterministic variants.

ISO 26262 requires an analysis of single- and multiple-point failures, and whether failures are detected or undetected. We extract this information from the causes in the fault trees. A cause representing a single-point failure contains a single failure event, other causes are multiple-point failures. For example, Cause 2 of variant 2 is a single-point failure since it contains the single failure event `ADS_backup_undetected`. An undetected failure is represented by a cause that contains at least one undetected failure event. Cause 6 of variant 1 represents such an undetected failure. With this information it is possible to perform further analyses on undetected failure rates and to relate them to single- and multiple-point faults, as required by ISO 26262.

7.5 Conclusion

We have presented an automated approach to support the design time functional safety analysis for architectures supporting ADS with Causality Checking. The presented case study addresses the handling of the complexity of future ADS by analyzing a flexible mapping of hardware and software functions. We have applied the proposed approach to two variants of a practical ADS architecture and compared the two variants. We have shown that the proposed approach gives the necessary information to perform functional safety analyses in the spirit of ISO 26262. The analysis included fail-operational behavior, software faults and interdependent driving functions which are so far not adequately addressed by ISO 26262. We see great potential in supporting ISO 26262 style functional safety analyses of innovative automotive architectures using the formal algorithmic analyses that QuantUM supports.

QuantUM is primarily an implementation of Causality Checking. Thus, this case study also shows that Causality Checking is of practical use to explain failures in untimed systems. Causality Checking analyzes discrete behavior and cannot deal with continuous behavior such as real-time. In the next part, we propose causal analyses for timed systems.

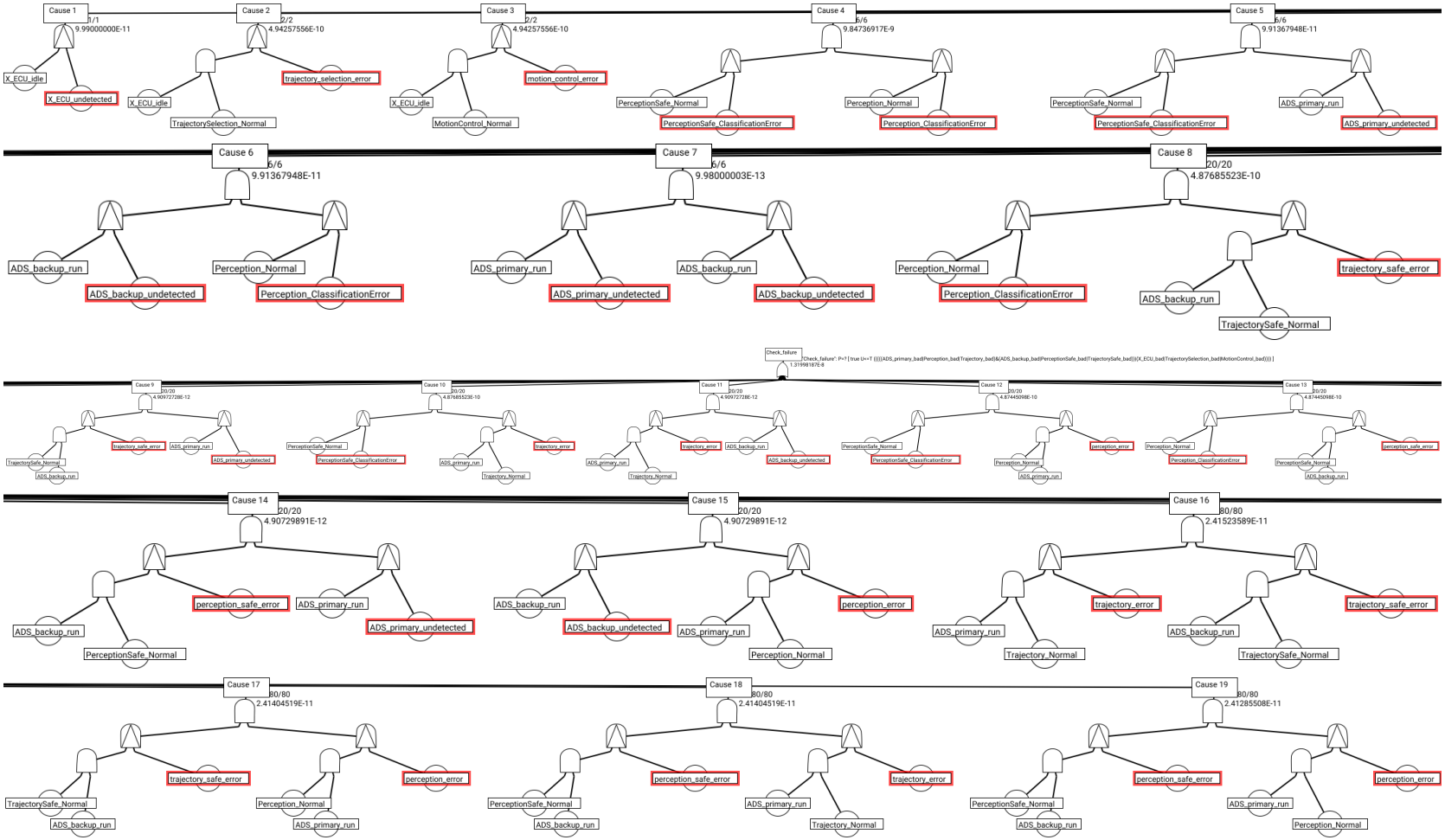


Figure 7.5: Fault Tree of Variant 1 (X_ECU)

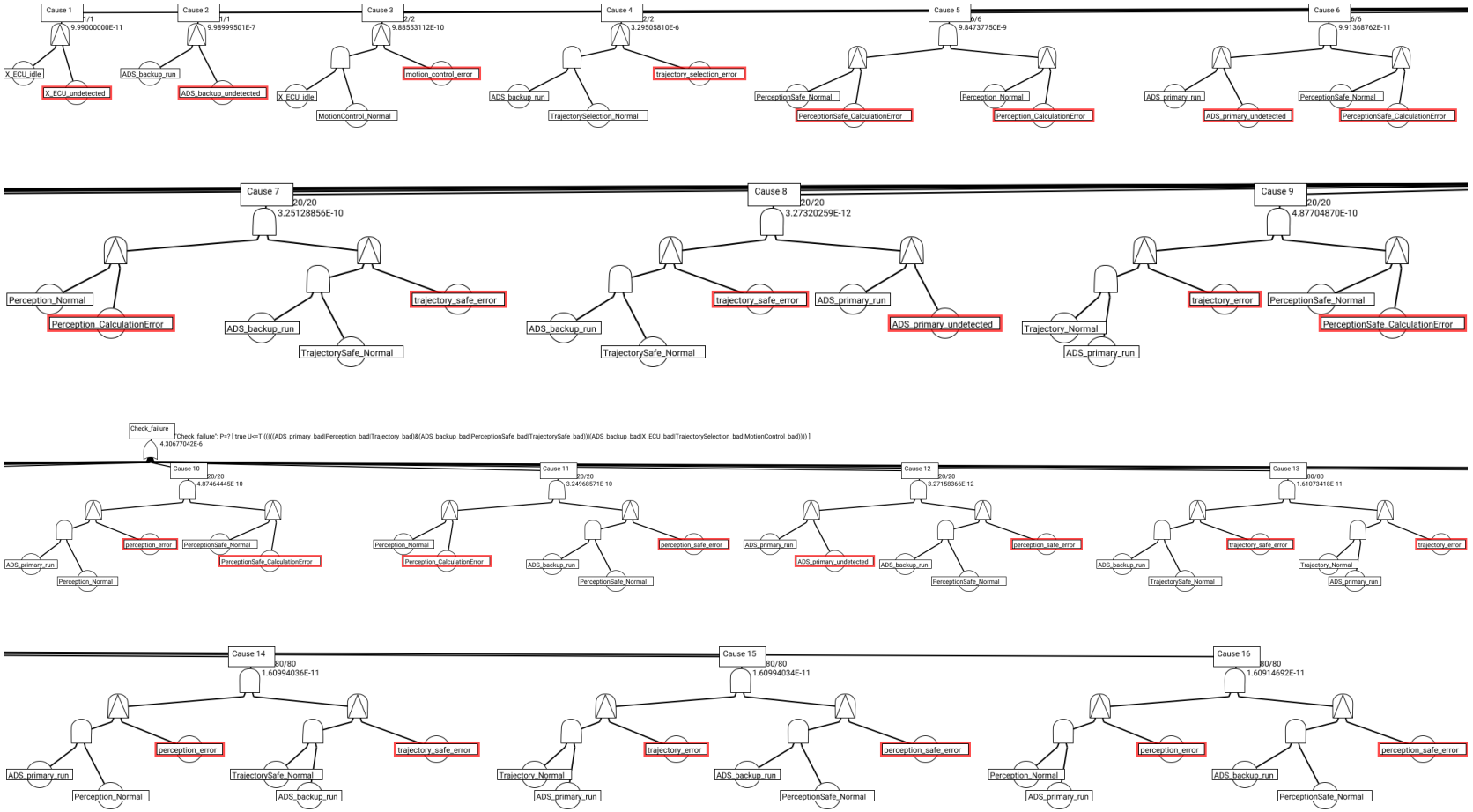


Figure 7.6: Fault Tree of Variant 2 (ADS_Star)

Part III

Causal Analyses and Repair Computation in Timed System Models

Chapter 8

Logical Encoding of Timed Diagnostic Traces

The content of this chapter is based on the publications [89, 91, 92].

8.1 Introduction

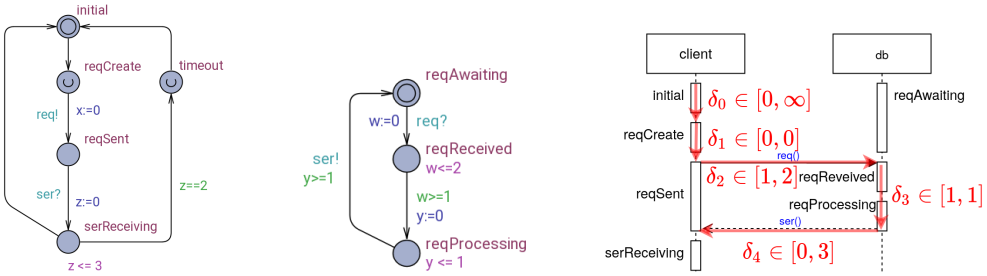
We are particularly interested in developing notions of causality and related analyses for models describing system computations. We have defined Causality Checking in Part II as a means to compute causes for the violation of reachability properties, relying on a counterfactual [108] notion of causality. The causes computed in that part rely on choices made during the dynamic execution of the model, for instance, a non-deterministically chosen interleaving of concurrent events during the execution of an untimed model, and we refer to this type of a cause as *dynamic causes*.

It is often sufficient to abstract from real time aspects when checking system properties, in particular when the focus is on functional aspects of the system as described in Part II. However, in certain domains, non-functional properties of the system such as response times or the timing of periodic behavior play an important role. In such cases, it is necessary to incorporate real time aspects into the models and the specification, as well as to use specialized real-time model checking tools, such as UPPAAL [26], Kronos [142], or opaal [39] during the verification step.

Next to the automatic nature of model checking, the ability to return counterexamples, in real-time model checking often referred to as timed diagnostic traces (TDT), is a further practical benefit of the use of model checking technology. A TDT describes a timed sequence of steps that lead the design model from the initial state of the system into a state violating a real-time property. However, just like a counterexample of an untimed model, a TDT itself is not a causal explanation of the property violation. The TDT also does not provide hints as to how to correct the model.

Since a TDT reaches a property violation, we assume in this part that the cause for the property violation is described by the timing in a TDT. The possible time delays are dynamically chosen and are constrained by syntactic features, such as clock constraints in transition guards. Hence, we propose a dynamic and a static analysis to compute causes for a TDT.

In Chapter 9, we examine the dynamics of a TDT with the counterfactual argument to compute causes for the violation of a timed reachability property. The time that a timed automaton spends in a certain location can be chosen non-deterministically, as long as it complies with the time constraints specified in the



(a) TA `client` (b) TA `db` (c) TDT illustrated using a time-interval annotated sequence diagram

Figure 8.1: Network of Timed Automata (NTA) - Database Example

model. The question, therefore, is whether there are time choices that can be considered causal for the violation of a desired timed reachability property.

In Chapter 10, we analyze the syntactic features of a TDT that constrain the possible time delays to compute syntactic repairs for the underlying timed automaton. Because a syntactic repair prevents the TDT that results in a property violation, it satisfies the counterfactual argument and we refer to it as a *static cause*.

Both analyses are based on the counterfactual argument and compare the alternative worlds that differ in the time choices or syntactic features of a TDT to find minimal sets of choices that lead to the effect. The analysis of both static and dynamic causes can help to identify possible modifications of a real-time model, establish safety cases for those types of models, or help with forensic system failure analysis.

The analyses in Chapters 9 and 10 rely on a logical formalization of a TDT that we present in the current chapter. Practical model checking tools generate TDTs in a textual format. We find TDTs using the tool UPPAAL [26] that stores a TDT in an XML file. In Section 8.2, we describe the components of a TDT and in Section 8.3, we develop a logical formalization of TDTs. We exemplify the logical encoding on the TDT of the real-time model in Example 3, which serves as a running example in the following. The TDT of this example describes the communication between a client and a database which in some cases is defective.

Example 3. *The database example is an NTA consisting of a TA* `client` *and a TA* `db` *depicted in Figure 8.1. The TDT that is depicted as a time annotated sequence diagram [24] in Figure 8.1(c), describes a request* `req` *of the* `client` *and the corresponding response* `ser` *of the* `db`*. The* `client` *and the* `db` *synchronize via the exchange of messages modeled by the pairs of send and receive actions* `req!` *and* `req?` *as well as* `ser!` *and* `ser?`*, respectively. The transmission time of the* `req` *message is controlled by the clock variable* `w` *and can range between 1 and 2 time units. This is achieved by the location invariant* `w <= 2` *on the* `reqReceived` *location in* `db` *together with the transition guard* `w >= 1` *on the transition from* `reqReceived` *to* `reqProcessing`*. A similar mechanism using clock variable* `z` *is used to constrain the timing of the transfer of message* `ser` *to be within 1 and 2 time units. The processing time in* `db` *is constrained to exactly 1 time unit by the location invariant*

`y <= 1` and the transition guard `y >= 1`. In `client`, a transition to location `timeout` can be triggered when the guard `z == 2` is satisfied in location `serReceiving`. The clock variable `x`, which is not reset until the next `req` message is sent, is recording the time that has elapsed since sending `req` and is used in location `serReceiving` in order to verify whether more than 4 time units have passed since `req` was sent. Every transition in the system is labeled with an action τ. The timed safety property that we will consider for our example is $\Pi = \neg@\texttt{client.serReceiving} \vee (x \leq 4)$. *For the violation of this property, UPPAAL produces the TDT* $S = \theta_0 \dots \theta_3$ *where*

$$\theta_0 = ((\mathit{initial}, \mathit{reqAwaiting}), \varnothing, \tau, \{w\}, (\mathit{reqCreate}, \mathit{reqAwaiting}))$$
$$\theta_1 = ((\mathit{reqCreate}, \mathit{reqAwaiting}), \varnothing, \tau, \{x\}, (\mathit{reqSent}, \mathit{reqReceived}))$$
$$\theta_2 = ((\mathit{reqSent}, \mathit{reqReceived}), \{w \geq 1\}, \tau, \{y\}, (\mathit{reqSent}, \mathit{reqProcessing}))$$
$$\theta_3 = ((\mathit{reqSent}, \mathit{reqProcessing}), \{y \geq 1\}, \tau, \{z\}, (\mathit{serReceiving}, \mathit{reqAwaiting})).$$

8.2 Timed Diagnostic Trace (TDT)

A TDT is a linear data structure that represents steps from an initial state of the system leading into a property violating state of an NTA. Given an NTA N and property Π, this means that the final state in the TDT is violating Π. Figure 8.2 represents an excerpt of a TDT obtained from the UPPAAL tool. The TDT illustrates the violation of the property $\Pi = \neg@\texttt{client.serReceiving} \vee (x \leq 4)$ by the NTA in Figure 8.1.

The TDT contains information regarding locations, location invariants, transition guards and reset operations affected by the execution of transitions. It also contains a representation of difference bound matrices (DBMs) that are attached to every location in the NTA to encode zones. The TDTs that we consider are symbolic in the sense that they do not give concrete time values at which certain locations are reached, but rather constraints on the clock values that need to be met for the property violating state to be reachable. For instance, consider the DBM table entry `<clockbound clock1="sys.x" clock2="sys.t(0)" bound="5" comp="<="/>` in Figure 8.2 which represents the constraint on x in the state `serReceiving` as $x \leq 5$. In other words, no concrete value of x is given when entering the violating state.

The objective of our analyses is either in Chapter 9 to identify delay assignments that result in a violation of a timed reachability property, or in Chapter 10 to propose repairs on the underlying NTA model. The symbolic representation of the valid clock assignments per location in the TDT given by DBMs, as done, for example, by UPPAAL, is not useful for this purpose. This is due to the fact that DBMs perform operations and optimizations on the constraints when they construct zones [27]. As a result, bound values in the UPPAAL code can not necessarily be directly associated with bound entries in the DBM. However, we need to use bound values to identify causes and to perform static repairs of the UPPAAL model. It is therefore necessary to define our own symbolic semantics of the TDT. This symbolic semantics is based on the following observations:

- The TDT represents an equivalence class of runs of the NTA in the sense

```
<trace initial_node="State1" trace_options="symbolic">
   <system>
      <clock name="t(0)" id="sys.t(0)"/>
      <clock name="x" id="sys.x"/>
      ...
      <process id="db" name="db">
         <location id="db.reqAwaiting" name="reqAwaiting">...</location>
         <location id="db.reqProcessing" name="reqProcessing">
             <![CDATA[1 && y <= 1]]>
         </location>
      ...
      <edge id="db.Edge3" from="db.Processing" to="db.reqAwaiting">
         <guard>y >= 1</guard>
         <sync>ser!</sync>
         <update>z:=0</update></edge>
      ...
      <edge id="client.Edge3" from="client.reqSent" to="client.serReceiving">
         <guard>1</guard>
            <sync>ser?</sync>
            <update>1</update></edge>
      ...
   </system>
   ...
   <node id="State4" location_vector="LocVec4" .../>
   <node id="State5" location_vector="LocVec5" .../>
   ...
   <location_vector id="LocVec4" locations="client.reqSent db.reqProcessing"/>
   <location_vector id="LocVec5" locations="client.serReceiving db.reqAwaiting"/>
   ...
   <dbm_instance id="DBM5">
      <clockbound clock1="sys.x" clock2="sys.t(0)" bound="5" comp="<="/>
   </dbm_instance>
   ...
   <transition from="State4" to="State5" edges="client.Edge3 db.Edge3 "/>
</trace>
```

Figure 8.2: Excerpt from the XML Representation of a Symbolic TDT Generated by UPPAAL

that all runs in this class traverse the same sequence of action transitions. This yields a sequence $l_0 ... l_n$ of NTA locations that are reached during the execution of the TDT.

- For every location l_j, there is a corresponding sojourn time in this location that we denote using a positive real valued delay variable δ_j.

- The value of every clock variable c in location l_j, which we denote by c_j, is constrained by the value of c_j when entering the location plus the delay δ_j, as well as, any location invariant constraint that refers to c in location l_j. Notice that we need to be able to refer to the values of clocks in every individual location, which is why we choose a Static Single Assignment form encoding of the clock variables.

- For any clock c in the successor location l_{j+1}, the value c_{j+1} is determined by the value c_j plus the sojourn time δ_j, if the clock is not being reset during the transition from l_j to l_{j+1}, or 0 otherwise.

- The sojourn time δ_j in a location l_j is constrained by the clock invariant conditions of the NTA component TAs that perform the computation step triggering the transition into location l_{j+1}. This means that the clock value c_j plus δ_j are bounded by any clock constraint that refers to a clock variable c. For instance, in the TDT in Figure 8.2, the location LocVec4 contains the TA location `reqProcessing` with the invariant `y <= 1`. Hence, the sojourn time in LocVec4 is constrained by $y_4 + \delta_4 \leq 1$.

- The transition from location l_j to location l_{j+1} is guarded by clock constraints on some of the clocks c, if at least one is given, and otherwise by *true*. This means that the time when entering location l_j as well as the time elapsing while in location l_j are constrained by these guards. We represent this by adding constraints of the form $c_j + \delta_j \sim \beta$ with $\beta \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$ to the symbolic constraint system. Notice that only the transition guards of the component TAs that performed the step leading to the step in the TDT need to be taken into account. In case the step performs a synchronization, the conjunctions of the transition guards for all clock variables on the sending and the receiving transitions need to be considered. To illustrate this point consider the `<transition from="State4" to="State5" edges="client.Edge3 db.Edge3"/>` entry in the TDT in Figure 8.2 which specifies that both the `client` and `db` component TAs are executing a step. The synchronization is indicated by the `<sync>ser!</sync>` and `<sync>ser?</sync>` entries which refer to edges in the transition graph of the TAs engaged in this synchronous communication step. The `<guard>1</guard>` and `<guard>y >= 1</guard>` entries in these edges imply that this transition needs to be guarded by the condition $true \wedge y \geq 1$, where the *true* corresponds to the transition guard in the `client` TA and $y \geq 1$ is the guard in the `db` TA.

- When a location l_j is labeled as `urgent`, l_j has to be left immediately and we constrain the delay δ_j in this location to have value 0.

8.3 TDT Formalization

Our analysis relies on a logical encoding of TDTs in the theory of quantifier-free linear real arithmetic. For the remainder of this chapter, we fix a TA $T = (L, l^0, C, \Sigma, \Theta, I)$ with a safety property Π and assume that $S = \theta_0, \ldots, \theta_{n-1}$ is an STT (see Definition 17) of T. We use the following notation for our logical encoding where $j \in [0, n+1]$ is a position in a realization of S and $c \in C$ is a clock:

- l_j denotes the location of the pre state of θ_j for $j < n$ and the location of the post state of θ_{j-1} for $j > 0$.

- c_j denotes the value of clock variable c when reaching the state at position j.
- δ_j denotes the delay of the delay transition leaving the state at position $j \leq n$.
- $reset_j$ denotes the set of clock variables that are reset by action θ_j for $j < n$.
- $ibounds(c, l)$ denotes the set of pairs $(\beta, \sim)$ such that the atomic clock constraint $c \sim \beta$ appears in the location invariant $I(l)$.
- $gbounds(c, \theta)$ denotes the set of pairs $(\beta, \sim)$ such that the atomic clock constraint $c \sim \beta$ appears in the guard of action θ.
- *urgent* denotes the set of indices of those locations l_j, $0 \leq j \leq n$, which are marked as *urgent* in the TA model from which S has been derived.

To illustrate the use of *ibounds*, assume location l to be labeled with invariants $x > 2 \wedge x \leq 4 \wedge y \leq 1$, then $ibounds(x, l) = \{(2, >), (4, \leq)\}$. The usage of *gbounds* is accordingly.

Definition 37. *The* timed diagnostic trace constraint system *(TDTCS) associated with STT S is the conjunction $\mathcal{T}$ of the following constraints:*

$$\mathcal{C}_0 \equiv \bigwedge_{c \in C} c_0 = 0 \qquad \textit{(clock initialization)}$$

$$\mathcal{A} \equiv \bigwedge_{j \in [0,n]} \delta_j \geq 0 \qquad \textit{(time advancement)}$$

$$\mathcal{R} \equiv \bigwedge_{c \in reset_j,} c_{j+1} = 0 \qquad \textit{(clock resets)}$$

$$\mathcal{D} \equiv \bigwedge_{c \notin reset_j} c_{j+1} = c_j + \delta_j \qquad \textit{(sojourn time)}$$

$$\mathcal{I} \equiv \bigwedge_{(\beta,\sim) \in ibounds(c,l_j)} c_j \sim \beta \wedge c_j + \delta_j \sim \beta \qquad \textit{(location invariants)}$$

$$\mathcal{G} \equiv \bigwedge_{(\beta,\sim) \in gbounds(c,\theta_j)} c_j + \delta_j \sim \beta \qquad \textit{(transition guards)}$$

$$\mathcal{U} \equiv \bigwedge_{j \in urgent} \delta_j = 0 \qquad \textit{(urgent location)}$$

$$\mathcal{L} \equiv @l_n \wedge \bigwedge_{l \neq l_n} \neg @l \qquad \textit{(location predicates)}$$

Let further $\Phi \equiv \Pi[\mathbf{c}_{n+1}/\mathbf{c}]$ be a property constraint where $\Pi[\mathbf{c}_{n+1}/\mathbf{c}]$ is obtained from Π by substituting every occurrence of a clock $c \in C$ by c_{n+1}. For instance, for the property Π used in Example 3, we have a $\Phi = \neg @client.serReceiving \vee x_5 < 4$. Then, the Π-extended TDTCS associated with S is defined as $\mathcal{T}^{\Pi} = \mathcal{T} \wedge \neg\Phi$.

A satisfying assignment of a TDTCS $\mathcal{T}$ is a variable assignment to every clock c_j and delay variable δ_j in $\mathcal{T}$ that satisfies every constraint in $\mathcal{T}$. A satisfying assignment of $\mathcal{T}$ induces a realization $\delta_0 ... \delta_n$ of S. We also denote a realization $\delta_0 ... \delta_n$ satisfying $\mathcal{T}$ by $\mathcal{T}[\delta_0 \dots \delta_n]$. The clock variables are syntactic sugar and can be removed from the TDTCS by replacing all occurrences of a clock variable c_j

with $\Sigma_{j' \leq i \leq j}\, \delta_i$ where $j' = 0$ or the index of the last transition with a reset of clock c before c_j.

In the analyses of a TDT, we use the following terminologies and properties referring to the concept of a TDTCS in Definition 37. A partial realization $\delta = \delta_0 \dots \delta_j$ of $\mathcal{T}$ with $0 \leq j \leq n$ is a realization of a TDTCS $\mathcal{T}_j$, where $\mathcal{T}_j$ encodes only the first j transitions of a given TDT. A suffix blocking partial realization $\delta'_0 \dots \delta'_j$ is a partial realization that satisfies $\mathcal{T}_j[\delta'_0 \dots \delta'_j]$ while $\mathcal{T}[\delta'_0 \dots \delta'_j]$ is unsatisfiable. A TDTCS is convex since it is a conjunction of constraints [120].

$\mathcal{T}$ is a correct formalization of S when any assignment of $\mathcal{T}$ contains a realization of S and any realization r in S is part of an assignment of $\mathcal{T}$.

Theorem 12. *When a TDTCS $\mathcal{T}$ is created for an STT S according to Definition 37, then $\delta^c_0, \dots, \delta^c_n$ is a realization of S iff there exists a satisfying variable assignment ι for $\mathcal{T}$ such that for all $j \in [0, n], \iota(\delta_j) = \delta^c_j$.*

Proof. Assume an STT $S = \theta_0, ..., \theta_{n-1}$ of a TA, a TDTCS $\mathcal{T}$ of S and an arbitrary delay sequence $\delta = \delta_0, \dots, \delta_n$. We need to prove that δ either satisfies S and $\mathcal{T}$ or none of both. We show by induction over the delays that after each delay δ_j the clocks values are equivalent in S and $\mathcal{T}$, and satisfy the same clock constraints.

Base Case: In the initial state, no delay has yet occurred. The value of every clock in S is 0, which is also ensured in $\mathcal{T}$ by $\mathcal{C}_0$.

Induction Step: It holds that the clock values in S and $\mathcal{T}$ are equivalent for $\delta_0 ... \delta_{j-1}$. We assume that after a time delay δ_j a transition θ_j transits from a location l_j to a location l_{j+1}. A constraint in S or $\mathcal{T}$ is not satisfied for δ_j in one of the following cases:

- When δ_j is negative, it violates the semantics of S but also violates the constraints in $\mathcal{A}$ of $\mathcal{T}$.
- After δ_j, a guard $c_j \sim \beta$ of a clock c_j is not enabled. The clock value of c_j was equivalent by assumption in S and $\mathcal{T}$ before δ_j, and so is equivalent after δ_j. Since the guard is by construction of $\mathcal{T}$ encoded in S and $\mathcal{T}$ and the clock value of c_j is equivalent, the guard is not enabled in S and $\mathcal{T}$.
- After δ_j, a clock c_j does not satisfy a location invariant $c_j \sim \beta$ in l_j. Since the invariant is by construction encoded in S and $\mathcal{T}$ and the clock value of c_j is equivalent as shown above, $c_j + \delta_j$ is violated in S and $\mathcal{T}$.
- A clock c_j does not satisfy a location invariant $c_j \sim \beta$ in l_{j+1}. By construction of $\mathcal{T}$, the invariant is encoded in S and $\mathcal{T}$. The clock value of c_j is either $c_j + \delta_j$ or 0 when c_j is reset. By the construction of $\mathcal{T}$, c_j is reset in $\mathcal{T}$ exactly when it is reset by θ_j in S. Thus, $c_j \sim \beta$ in l_{j+1} is not satisfied in S and $\mathcal{T}$.
- δ_j is not 0, although l_j is an urgent location. By construction of $\mathcal{T}$, when l_j is in $\mathcal{U}$ then only $\delta_j = 0$ is an assignment of $\mathcal{T}$. Thus, S and $\mathcal{T}$ are not satisfied.
- With any other value of δ_j, the time constraints in S and $\mathcal{T}$ are satisfied.

As we see, the clock values in S and $\mathcal{T}$ are equivalent after each delay, and also equivalently satisfy the clock constraints in S and $\mathcal{T}$. In consequence, δ satisfies S and $\mathcal{T}$, or none of both. □

Theorem 12 ensures that an assignment of $\mathcal{T}$ and the contained realization of S satisfy the same time constraints and location predicates. Consequently, they also equivalently violate or satisfy the time constraints and location predicates of which an invariant property Π exists. This allows us to infer Corollary 1 because the conjunction of $\mathcal{T}$ and Π is a Π-extended TDTCS $\mathcal{T}^{\Pi}$. A modification of $\mathcal{T}^{\Pi}$ that turns the formula unsatisfiable is also a potential repair in the underlying TA since it prevents the property violation. We will use this insight in Chapter 10 to compute a repair.

Corollary 1. *An STT S witnesses a violation of Π in a TA iff $\mathcal{T}^{\Pi}$ is satisfiable.*

Proof. We know by Theorem 12 that a realization r of an STT exists iff a satisfying assignment ι of the STT exists. r and ι satisfy equivalent location predicates and clock constraints and, thus, r and ι equivalently hold or violate Π. □

Based on the TDTCS encoding in Definition 37, in the next chapters, we present causal analyses to explain the occurrence of a TDT and also to repair the underlying timed automaton.

Chapter 9

Causal Ranges for the Violation of Timed Properties

The content of this part is based on the publication [89].

Contents

9.1 Introduction

We now illustrate our idea of a dynamic cause for a TDT, based on the TDT of the database in Example 3. A TDT is symbolic in that it describes a set of executions where the time delays before taking the next transition are represented by symbolic variables and constrained by symbolic constraints. The TDT of the database example and the possible time delays are depicted by the sequence diagram in Figure 8.1(c). A concrete assignment of the delay values $\delta_0 ... \delta_4$ is a realization, and represents the real-time characteristics of a concrete execution of the TDT. A realization may, or may not, violate the considered property. An assignment in which the minimal possible values are assigned to all δ_js, in other words, the realization $\delta_0 = 0$, $\delta_1 = 0$, $\delta_2 = 1$, $\delta_3 = 1$ and $\delta_4 = 0$, leads to a trace that does not violate the considered property. Notice that the values of δ_1 and δ_3 are fixed. The values of δ_0, δ_2 and δ_4 in a concrete execution may be determined by environmental effects, or by non-deterministic internal decisions of the two involved subsystems, for instance, as a result of task scheduling or memory management. If we consider an alternate execution in which $\delta_4 = 3$, while all other delay values remain as described above, then this execution violates the considered property.

Some assignments of values to the delays satisfy the property, while others violate the property. This indicates that the cause for the property violation is to be found in the value assignment of certain delays. We are interested in characterizing value assignments to the δ_is that inevitably lead to a property violation using linear constraints. We base the analysis on counterfactual causal reasoning [63] and call

these constraints causal ranges. The causal ranges for the TDT in Example 3 are $2 \leq \delta_4 \leq 3$ and $3 \leq \delta_2 + \delta_4 \leq 5$.

In this chapter, we present a formalization of causal ranges and an algorithm to compute causal ranges for a TDT. The evaluation of the algorithm shows that a causal range can explain the violation of a property.

Structure of the Chapter. We derive the notion of a causal range from causal reasoning in Section 9.2. In Section 9.3, we present a formal framework of dynamic trace analysis for causal delays and causal ranges, and present an algorithm to compute causal ranges in Section 9.4. We evaluate an implementation of the algorithm in the tool CaTiRA in Section 9.5 by computing causal ranges for several timed automata models. Finally, we draw conclusions in Section 9.6.

9.2 Causal Reasoning for Time Delays in a TDT

We now motivate our definition of a causal range and exemplify our proposed causal analysis. A TDT contains all information regarding possible assignments of the delay variables, in particular, the constraints determining possible value ranges for these assignments. As discussed above, these value assignments in a realization of a TDT determine whether a property is violated.

Again, we use the counterfactual argument [108] to establish when certain assignments of delay variables constitute a cause for a property violation. In this chapter, we formalize a cause by following conditions and say that one phenomenon is a *cause* of another phenomenon, called an *effect*, if and only if

I whenever the cause applies, the effect is observed (regularity argument),

II when the cause does not apply, the effect will not be observed either (counterfactual argument), and

III no true subset of the cause ensures I and II (minimality argument).

In order to establish a causal relation between an assignment of values to the delay variables and the violation of a timed safety property, we develop criteria for what we understand to be a cause. For a TA, a dynamic cause is a constraint on delay assignments where every assignment that satisfies the cause violates a given property (condition I). Our interpretation of II is that several independent causes can result in a property violation but at least one assignment exists that is not violating the property. III is a minimality argument and removes from a cause any constraint that has no influence on whether an assignment violates the property or not.

Applying this reasoning to Example 3, choosing either $2 \leq \delta_4 \leq 3$ or $3 \leq \delta_2 + \delta_4 \leq 5$, with arbitrary but admissible values assigned to all other δ_js, means that the desired property is violated. In addition, a different choice of the values exists that does not satisfy one of these two expressions and the property violation is not observed. We conclude that, following the counterfactual argument, these two constraint expressions are to be considered independent causes for the property violation.

We now illustrate the computation of a cause in the form of a *causal range* for the TDT in Example 3. For a range expression, to be causal, the values for all δ_js in this range have to violate the considered property (I). Furthermore, it needs to satisfy the counterfactual argument (II) which means that there is a different assignment of values to at least one of the δ_js such that this assignment violates the range constraint and does not lead to a property violation. This means that in order to check whether II is satisfied overall, we can test II on every δ_j in isolation. To illustrate this point, consider the realization $\delta_0 = 1$, $\delta_1 = 0$, $\delta_2 = 1.5$, $\delta_3 = 1$, and $\delta_4 = 1.5$. The realization also violates the considered property with any other value for δ_0, while a decrease of the assigned values of δ_2 or δ_4 results in a realization that satisfies the property. The assignment of δ_0 has no impact on the property since its satisfaction solely depends on the values of δ_2, δ_3 and δ_4. The values of δ_1 and δ_3 are fixed and, hence, they have no admissible alternative assignment. We conclude that only δ_2 and δ_4 have the potential to prevent the violation of the property and, hence, satisfy II. We, therefore, call them causal delay variables. Next, we determine the range in which δ_2 and δ_4 have realizations that violate the considered property.

- A realization with an assignment of δ_4 in the range $[0, 1[$ never violates the property. On the other hand, any realization with an assignment of δ_4 in the range $[2, 3]$ leads to a violation. For every of these realizations, a realization exists that is identical, except for the value of δ_4, that does not violate the property. This means that I and II are satisfied and since δ_4 is the only delay in the range (III), we conclude that the constraint $2 \leq \delta_4 \leq 3$ represents a causal range.

- For any realization with an assignment of δ_4 in the range $[1, 2[$, the violation of the property depends on the assignment of the delay variable δ_2. Hence, we analyze which values can be assigned to δ_2 so that this assignment is admissible according to the constraints in the TDT. We then detect that for any admissible assignment of δ_2 in the TDT of Example 3, it conversely depends on the assignment of δ_4 whether a realization violates the property. Thus, δ_2 and δ_4 need to jointly be considered when determining causal ranges. In order to specify the interdependence of δ_2 and δ_4, we consider their sum. We, hence, analyze the range of assigned values for which the sum of δ_2 and δ_4 violates the property. We see that any realization satisfying $3 \leq \delta_2 + \delta_4 \leq 5$ violates the property, a realization not in this range exists that satisfies the property and no causal range exists for a subset of the delays that describes the realization satisfying $3 \leq \delta_2 + \delta_4 \leq 5$. This means that I, II and III are satisfied and $3 \leq \delta_2 + \delta_4 \leq 5$ is a second causal range.

In the sequel, we will present an algorithmic way of determining the constraints describing causal ranges.

9.3 Formalizations to Compute Causal Ranges

The aim of this section is to define a causal range as a range of values for a given delay set D where any value satisfies the regularity argument (I), the counterfactual

argument (II) and the minimality argument (III).

Before we define a causal range, we need to determine a delay set D. We introduce the notion of a *delay set*, which is a subset of the indices of the delay variables $\delta_0, \ldots, \delta_n$ occurring in the STT. For instance, the set $\{\delta_2, \delta_4\}$ is represented by the delay set $\{2, 4\}$. Any delay variable δ_j in D shall be causal, which means that a realization δ exists where the value assignment of δ_j determines whether δ violates or satisfies the property constraint Φ of a given timed safety property. In case δ exists, we call δ_j a *causal delay variable*. Whether δ satisfies or violates Φ can depend on the value assignment of several causal delay variables, as we have seen before. In this case, a realization δ exists such that a different value assignment of any δ_j in D can results in a realization that satisfies the property. We formally define the existence of δ for a delay set D in Definition 38. CV1 in Definition 38 ensures that a realization δ exists for a value v where the sum of the delay assignments with an index in D is equivalent to $v = \sum_{j \in D} \delta_j$. This δ violates the property constraint Φ, and consequently satisfies the regularity argument (I). CV2 ensures II by requiring the existence of an alternate assignment for every delay variable with an index in D, resulting in a realization δ' that does not violate the property. δ' can also be a *suffix-blocking partial realization* $\delta_0 \ldots \delta_{j-1}\delta'_j$ which cannot be completed to a full realization in a way that would violate the property. For instance, consider a TDT with a guard on a transition that leads to an immediate property violation and where the guard is enabled for an assignment $\delta_j < 2$ and disabled for an assignment $\delta_j \geq 2$. Thus, assigning $\delta_j = 2$ prevents the property violation to be reachable. In conclusion, a causal value satisfies I and II for a realization δ, and witnesses that any δ_j in D is a causal delay variable.

Definition 38 (Causal Value)**.** *Assume a TDTCS $\mathcal{T}$ for a TDT of length n, a delay set D of $\mathcal{T}$ and a property constraint Φ. A causal value is a value v in a delay set constraint $v = \sum_{j \in D} \delta_j$ where $\delta_j \in \mathbb{R}_0^+$ that satisfies:*

CV1 *There exists a realization $\delta = \delta_0 ... \delta_n$ with delay values $\delta_j \in \mathbb{R}_0^+$ for $0 \leq j \leq n$ that satisfies $v = \sum_{j \in D} \delta_j$ and violates Φ.*

CV2 *For every delay value δ_j with $j \in D$, a different delay value δ'_j with $\delta_j \neq \delta'_j$ exists, so that either $\delta_0 \ldots \delta_{j-1}\delta'_j$ is a suffix-blocking partial realization or $\delta_0 \ldots \delta_{j-1}\delta'_j\delta_{j+1} \ldots \delta_n$ satisfies $\mathcal{T}$ and Φ.*

In the TDT in the example from Figure 8.1(c), a causal value of the delay set $\{4\}$ is every value in the range $[1, 3]$, of the delay set $\{2\}$ is every value in the range $[1, 2]$ and of the delay set $\{2, 4\}$ is every value in the range $[3, 4]$.

Not all values from a causal range are causal values. For instance, in the TDT in Figure 8.1(c), the value $v = 5$ is part of the causal range $3 \leq \delta_2 + \delta_4 \leq 5$, and has a realization with the assignments $\delta_2 = 2$ and $\delta_4 = 3$ that violates the property Φ, thus satisfying conditions CV1 and I. CV2 requires that an alternative assignment for each of δ_2 and δ_4 exists that prevents the property violation. Such an assignment does not exist for δ_2 but for δ_4 with the assignment $\delta_4 = 0$. Thus, $v = 5$ is not a causal value since it violates CV2, even though it satisfies II. The purpose of a causal value is different from a causal range. A causal value ensures that the assignment of every delay variable in D influences for at least one realization whether the realization

violates the property. In difference, a causal range is a cause and ensures that for every realization that satisfies the cause a different assignment exists that satisfies the property.

Next, we define a causal range for a delay set D in Definition 39. A causal range is a constraint of the form $l \sim \sum_{j \in D} \delta_j \sim u$ with a lower bound l and an upper bound u. Every value v in a causal range has to satisfy the causal arguments I, II and III. Condition CR1 in Definition 39 claims that for every value v in the causal range, a realization exists with $v = \sum_{j \in D} \delta_j$. CR2 ensures the regularity argument I that any realization which satisfies the causal range violates the property. Let $\max(D)$ return the maximal value of all elements in D. A partial realization $\delta_0 \dots \delta_m$ can also satisfy causal range constraints since a value is assigned to every delay in D for $m \geq \max(D)$. In order to satisfy causal condition I, this partial realization is not allowed to be a suffix-blocking partial realization that prevents the property violation. CR2 refers to all partial realizations. Notice that, in particular, a partial realization with $m = n$ satisfies $\mathcal{T}$, and violates the property Φ. We conclude that any realization that satisfies the causal range constraint violates the property, and CR2 actually ensures I. Condition CR3 ensures that a causal value v_c in the range exists that is not a causal value for a true subset of D. We interpret the causal minimality argument III such that we require the number of delay variables that are part of a causal range to be minimal. When every v_c is already part of another causal range r' for a true subset of the variables in D, then we conclude that r' is a more concise cause for the violation of property Φ which violates III. The existence of v_c, as required by CR3, is necessary to ensure II, as shown in the proof of Theorem 13. Condition CR4 claims that the causal range r is maximal in the sense that there is no truly larger range encompassing r which satisfies CR1 to CR3. We include this constraint based on the assumption that an analysis result consisting of fewer causal ranges is easier to interpret than one with more causal ranges.

Definition 39 (Causal Range)**.** *Assume a TDTCS $\mathcal{T}$ for a TDT of length n and a property constraint Φ. A causal range r is a constraint for a delay set D of the form $l \sim \sum_{j \in D} \delta_j \sim u$ and $\sim \in \{<, \leq\}$, where $l, u \in \mathbb{R}_{\geq 0}$, delay values $\delta_j \in \mathbb{R}_0^+$ and the following conditions hold:*

CR1 *For every value $v \in [l, u]$, exists a realization $\delta_0 \dots \delta_n$ with $v = \sum_{j \in D} \delta_j$.*

CR2 *For every partial realization $\delta_0 \dots \delta_m$ where $max(D) \leq m \leq n$ and the value $v = \sum_{j \in D} \delta_j$ is in $[l, u]$, there exists a realization $\delta_0 \dots \delta_m \dots \delta_n$ that violates Φ.*

CR3 *At least one value v_c in the causal range is a causal value of D, and v_c is not a causal value for a true subset of D.*

CR4 *The range r is maximal, that means any lower bound $l' < l$ and any higher bound $u' > u$ will not satisfy CR1 to CR3.*

As an example, the constraint $2 \leq \delta_4 \leq 3$ and the constraint $3 \leq \delta_2 + \delta_4 \leq 5$ satisfy the definition of a causal range for the example in Figure 8.1(c).

A causal range is only a cause when it satisfies the counterfactual argument (II) and is ensured by Theorem 13. CR2 ensures that every realization r with value v

violates a given property Φ. The counterfactual argument is satisfied for v when for every r, a different value assignment only of the delay variables in D exists such that the resulting realization satisfies Φ.

Theorem 13. *For every value in a causal range, the counterfactual argument (II) is satisfied.*

Proof. Assume a causal range r with a delay set D that satisfies conditions CR1 to CR4 in Definition 39 and an arbitrary value v in r. Condition CR3 ensures that a realization δ' for a causal value exists that satisfies the property. We now prove that a realization δ with value v exists for which δ' witnesses II. δ' has a value v' and witnesses that an assignment of the delay variables with index in D exists where the sum is v'. Condition CR1 ensures that a realization exists where the sum of the delay assignment with index in D is v. Since a delay assignment for v' and v exists and the TDTCS is convex, an assignment of the delay variables with index in D exists for every value between v' and v. II allows changing the assignment of delays with an index in D to another admissible assignment. We now either continuously increase or continuously decrease the assignment of a delay with an index in D for δ' until we arrive at a realization δ that has value v. δ has a value v in the causal range and since condition CR2 holds, δ violates the property. In summary, a δ with the value v that violates the property Φ has to exist, and a different assignment of delays with an index in D can result in δ'. This proves II. □

We have argued above that CR2 ensures the regularity argument I and CR3 ensures the minimality argument III. In combination with Theorem 13, we conclude that a causal range represents a cause for the property violation according to the definition in Section 9.2.

9.4 Causal Range Algorithm

We present the `Causal Range Algorithm` to compute a set of causal ranges for a given TDT S and a given property Φ. The input of the algorithm is a TDTCS $\mathcal{T}$ derived from S and a property constraint Φ created for the considered property. The output of the algorithm is a set of causal ranges, where any causal range is characterized by a real valued lower bound, a real valued upper bound and a delay set taken from the power set of the TDT delay variables. The algorithm performs the search for causal ranges by solving three satisfiability problems. These problems are depicted in the control flow diagram given in Figure 9.1. By solving the problem $P1$ the algorithm starts to iteratively compute the causal delay variables for $\mathcal{T}$. For every computed causal delay variable it creates a delay set. When $P1$ is no longer satisfiable, then the algorithm computes for every delay set D a causal range by solving the problem $P2$. After the range computation of a delay

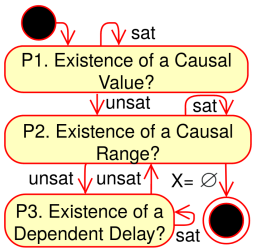


Figure 9.1: Control Flow Diagram of Causal Range Algorithm

set D becomes unsatisfiable, the algorithm solves problem $P3$ to check whether another causal delay variable δ_k causally depends on D, which is the case when a causal value exists for $D \cup \{k\}$. In case δ_k exists, the algorithm found a new delay set $D' = D \cup \{k\}$. The algorithm solves $P2$ and $P3$ for every delay set. We encode the problems $P1$ to $P3$ in linear real arithmetic as follows:

Problem P1: Existence of a Causal Value. The algorithm first checks for every delay variable δ_j in the TDT whether it is a causal delay variable. A delay variable δ_j is causal when a causal value for $D = \{\delta_j\}$ exists. We encode the conditions that CV1 and CV2 in Definition 38 specify for a causal value v in the constraint $\mathcal{C}_j$ (9.1). The constraint ensures that a realization $\delta = \delta_0, \ldots, \delta_n$ with δ_j and value $v = \delta_j$ exists. δ is a realization when it satisfies $\mathcal{T}$, and has to violate Φ in order to satisfy CV1 in Definition 38. The constraint $\mathcal{C}_j$ also ensures that CV2 is satisfied. It is satisfiable when an assignment δ'_j that is different from δ_j exists, such that either $\delta' = \delta_0 \ldots \delta_{j-1}\delta'_j$ is a suffix-blocking partial realization, or $\delta[\delta_j/\delta'_j]$ is a realization that satisfies Φ. When δ' is a suffix-blocking partial realization, it satisfies $\mathcal{T}^*$ (9.2). $\mathcal{T}^*$ ensures that δ' satisfies the constraint $\mathcal{T}_j$ for a partial realization and no assignment of δ'_{j+1} to δ'_n exists such that $\delta_0 \ldots \delta_{j-1}\delta'_j\delta'_{j+1}, \ldots, \delta'_n$ is a realization that satisfies $\mathcal{T}$.

$$\mathcal{C}_j \equiv (\exists \delta_0, \ldots, \delta_n, \delta'_j)(\mathcal{T} \wedge \neg\Phi \wedge (\mathcal{T}^*[\delta_0 \ldots \delta_{j-1}\delta'_j] \vee (\mathcal{T}[\delta_j/\delta'_j] \wedge \Phi[\delta_j/\delta'_j]))) \quad (9.1)$$

$$\mathcal{T}^*[\delta_0 \ldots \delta_{j-1}\delta'_j] \equiv \mathcal{T}_j[\delta_0 \ldots \delta_{j-1}\delta'_j] \wedge \neg(\exists \delta'_{j+1}, \ldots \delta'_n)(\mathcal{T}[\delta_0 \ldots \delta_{j-1}\delta'_j \ldots \delta'_n]) \quad (9.2)$$

By iteratively determining the satisfiability of the constraint $\mathcal{C}_j$ for all delay variables δ_j occurring in $\mathcal{T}$, the algorithm checks whether these δ_js actually are causal delay variables. In the implementation of the algorithm, we combine all $\mathcal{C}_j$ into one linear constraint system $\mathcal{C}$ of the form $\bigwedge_{0 \leq j \leq n} \neg c_j \vee C_j$ in which we add a fresh Boolean variable c_j for every occurring δ_j. These c_j are defined such that if some $\mathcal{C}_j$ is unsatisfiable, then $\neg c_j$ holds. We use a MaxSMT solver in order to determine the minimum number of c_j = true assertions that need to be violated in order to render $\mathcal{C}$ satisfiable. A delay variable δ_j is causal if and only if c_j is true in the solution to the MaxSMT problem. Subsequently, for every computed causal delay variable δ_j the algorithm adds a delay set $\{\delta_j\}$ to a first-in-first-out queue X. This queue stores every computed delay set and will be handed over to the algorithm addressing problem P2, which computes a causal range for every element of X. For the example in Figure 8.1(c), the queue passed on to P2 is $X = \langle\{4\}, \{2\}\rangle$.

Problem P2: Existence of a Causal Range. The algorithm solving problem P2 removes a delay set D from queue X and computes the causal ranges of D. We encode the computation of a causal range as a satisfiability problem. We formalize the conditions CR1 to CR3 in Definition 39 as individual constraints $\mathcal{R}_1$ to $\mathcal{R}_3$. The satisfiability of this conjunction yields an answer to problem P2 and computes causal ranges if they exist. CR1 claims that for every value t in the causal range $[l, u]$, a realization exists. If $\mathcal{R}_1^b$ is satisfiable, then there exists a realization of $\mathcal{T}$ with a value $b = \sum_{j \in D} \delta_j$. We now check whether $\mathcal{R}_1^b[b/l]$ and $\mathcal{R}_1^b[b/u]$ are satisfiable,

respectively. If both are satisfiable, due to the convexity of $\mathcal{T}$ we can conclude that $\mathcal{R}_1^b[b/t]$ is satisfiable for any value $t \in [l, u]$.

$$\mathcal{R}_1^b \equiv (\exists \delta_0 \dots \delta_n)(\mathcal{T} \wedge b = \textstyle\sum_{j \in D} \delta_j) \tag{9.3}$$

CR2 claims that for every partial realization $\delta_0 \dots \delta_j$ which satisfies $l \leq \sum_{j \in D} \delta_j \leq u$ there exists $\delta_0 \dots \delta_j \delta_{j+1} \dots \delta_n$ that violates the property Φ. We formalize CR2 as follows:

$$\begin{aligned} \mathcal{R}_2 \equiv \textstyle\bigwedge_{max(D) \leq j \leq n} (\forall \delta_0 \dots \delta_j)(\mathcal{T}_j[\delta_0 \dots \delta_j] \wedge (l \leq \textstyle\sum_{j \in D} \delta_j \leq u) \\ \Rightarrow (\exists \delta_{j+1} \dots \delta_n)(\mathcal{T} \wedge \neg \Phi)) \end{aligned} \tag{9.4}$$

We project a realization $\delta = \delta_0 \dots \delta_n$ to a single value v with the formula $v = \sum_{j \in D} \delta_j$. If $v \in [l, u]$, we say that δ is contained in $[l, u]$. Constraint $\mathcal{R}_3$ (9.5) ensures that a realization δ contained in $[l, u]$ exists and δ is not contained in any causal range $[l_i, u_i]$ of a subset D_i of D. When this δ exists, CR3 is fulfilled. Notice that the subset D_i is not necessarily a true subset of D and therefore ignores previously found causal ranges contained in D. If further causal ranges exist for D, then these causal ranges will also be computed by the algorithm solving P2.

$$\mathcal{R}_3 \equiv \exists \delta_0 \dots \delta_n . \mathcal{T} \wedge l \leq \sum_{j \in D} \delta_j \leq u \wedge \bigwedge_{D_i \subseteq D} (\sum_{i \in D_i} \delta_i < l_i) \vee (u_i < \sum_{i \in D_i} \delta_i) \tag{9.5}$$

A satisfying assignment for the conjunction of $\mathcal{R}_1^l$, $\mathcal{R}_1^u$, $\mathcal{R}_2$, and $\mathcal{R}_3$ contains a causal range $[l, u]$ which is not necessarily maximal. The algorithm takes advantage of the optimization possibilities of the SMT solver Z3 [42] to minimize l and to maximize u in the conjunction in order to ensure a maximal range (c.f. CR4). In the example of Figure 8.1(c) for the delay set $D = \{4\}$, the algorithm computes the causal range $2 \leq \delta_4 \leq 3$. Since no further causal range is found, then the algorithm proceeds with extending D by solving problem P3.

Problem P3: Existence of a Dependent Delay. A delay variable δ_k is dependent on a delay set D when a causal value exists for the delay set $D' = D \cup \{k\}$. The algorithm solving P3 extends a delay set D with the index of a delay variable δ_k depending on D. We encode the question whether a causal value for D' exists as the problem whether constraint $\mathcal{C}_k$ (9.6) is satisfiable. The satisfiability of $\mathcal{C}_k$ yields an answer to problem P3. A satisfying model then yields the dependent delays δ_k, if any exist. $\mathcal{C}_k$ ensures that there exists a realization $\delta = \delta_0 \dots \delta_n$ with value v that violates the property Φ. For every index j in the delay set D', a realization exists that differs from δ only in the assignment of δ_j and does not violate Φ, thus satisfying CV2 in Definition 38.

$$\mathcal{C}_k \equiv \exists \delta_0, \dots, \delta_n . \mathcal{T} \wedge \neg \Phi \wedge \bigwedge_{j \in D'} \exists \delta_j' . \mathcal{T}^*[\delta_0 \dots \delta_{j-1} \delta_j'] \vee (\mathcal{T}[\delta_j / \delta_j'] \wedge \Phi[\delta_j / \delta_j']) \tag{9.6}$$

For every causal delay variable δ_k with $k \notin D$ for which $\mathcal{C}_k$ is satisfiable, the algorithm adds $D' = D \cup \{k\}$ to the queue X. To illustrate this step, for the TDT in Figure 8.1(c) and the delay set $D = \{4\}$, $\mathcal{C}_k$ is satisfiable for $k = 2$ and the queue $X = \langle \{2\} \rangle$ is extended to $X = \langle \{2\}, \{4, 2\} \rangle$. The algorithm proceeds with removing the next delay set from X, solving problems P2 and P3 for this delay set, and terminates when X is empty.

9.4.1 Correctness of the Algorithm

The theorems below show that the algorithm to compute causal ranges is correct with respect to soundness and completeness.

Theorem 14 (Soundness of Causal Range Computation)**.** *Every causal range returned by the Causal Range Algorithm for a TDTCS $\mathcal{T}$ and a delay set D satisfies CR1 to CR4.*

Proof. Assume a causal range $[l, u]$ returned by the Causal Range Algorithm for a TDTCS $\mathcal{T}$ and a delay set D that is not satisfying one of the conditions CR1 to CR4. The algorithm returns $[l, u]$ for a satisfying assignment of the conjunction of $\mathcal{R}_1^l$, $\mathcal{R}_1^u$, $\mathcal{R}_2$ and $\mathcal{R}_3$.

A first case to consider is that CR1 is not satisfied. A violation of CR1 means that no realization exists for a value $v \in [l, u]$. By checking satisfiability of $\mathcal{R}_1^l$ and $\mathcal{R}_1^u$ the algorithm ensures that realizations δ_l with $l = \sum_{j \in D} \delta_j$ and δ_u with $u = \sum_{j \in D} \delta_j$ exist. The existence of these realizations combined with the convexity of $\mathcal{T}$ imply that a delay assignment exists for every value in $[l, u]$. Thus, a realization for every value in the causal range has to exist and CR1 cannot be violated.

As a second case, consider that CR2 is not satisfied. Thus, a suffix-blocking partial realization or a realization with a value v in $[l, u]$ exists that satisfies the property. This (partial) realization contradicts that $[l, u]$ satisfies $\mathcal{R}_2$. We see that CR2 has to hold.

The third case to consider is that CR3 is violated. As a consequence, $[l, u]$ contains no causal value or only causal values that are contained in causal ranges of true subsets of D. This contradicts the assumption that $\mathcal{R}_3$ is satisfied and thus, CR3 holds.

In the fourth case to consider is that CR4 is violated, since $[l, u]$ is not maximal. Thus, either a smaller lower bound l' exists with a causal range $[l', u]$ or an upper bound with a higher value u' exists with a causal range $[l, u']$. Both extended ranges would contradict that the algorithm uses optimization and returns the minimal lower bound and maximal upper bound. Thus, l' and u' cannot exist.

In all four cases, the causal range $[l, u]$ has to satisfy the conditions CR1 to CR4 of a causal range, and this contradicts the assumption that one of the conditions does not hold. □

Theorem 15 (Completeness of Causal Range Computation)**.** *The Causal Range Algorithm returns every causal range for any given TDT.*

Proof. Assume a causal range $[l, u]$ with a delay set D that is not returned by the Causal Range Algorithm for a TDTCS $\mathcal{T}$. This means that it either misses to create the delay set D, or misses to compute $[l, u]$ for D.

Consider first the case that the algorithm misses a delay set D. D consists either only of one causal delay variable δ_j and would be found since $\mathcal{C}_j$ is satisfiable, or D contains several causal delay variables. A D with several causal delay variables is found by the algorithm, when $\mathcal{C}_k$ is satisfiable for every subset D' of D. Since CR3 holds for the causal range $[l, u]$, a causal value v in $[l, u]$ has to exist. v witnesses that $\mathcal{C}_k$ is satisfiable for D. D is only found by the algorithm when a subset of D

with one delay less is found. Notice that the constraint $\mathcal{C}'_k$ for a subset D' of D, is a conjunction of a subset of the constraints in $\mathcal{C}_k$. Thus, $\mathcal{C}'_k$ is a relaxation of $\mathcal{C}_k$. Since $\mathcal{C}_k$ is satisfiable, $\mathcal{C}'_k$ is satisfiable for every subset D' of D. We see D has to be found by the algorithm, which establishes a contradiction to the assumption.

Next, consider the only alternative case that D is created by the algorithm, but $[l, u]$ is not found for D. By assumption, the causal range $[l, u]$ satisfies CR1 to CR4 in Definition 39. The algorithm does not compute $[l, u]$ when one of the constraints $\mathcal{R}_1^l$, $\mathcal{R}_1^u$, $\mathcal{R}_2$ or $\mathcal{R}_3$ is not satisfiable for $[l, u]$. $\mathcal{R}_1^l$, $\mathcal{R}_1^u$ are satisfiable otherwise no realization exists with value l or u and this would violate CR1. When $\mathcal{R}_2$ is unsatisfiable, a partial realization $\delta_0 \dots \delta_j$ with a value in the range $[l, u]$ exists that is either a suffix-blocking partial realization or every extension $\delta_0 \dots \delta_j \delta_n$ satisfies the property. In both cases, $\delta_0 \dots \delta_j$ violates CR2. Alternatively, we assume $\mathcal{R}_3$ is unsatisfiable. In this case, any value in $[l, u]$ is contained in a causal range of a subset of D but this violates CR3. Finally, the Causal Range Algorithm minimizes l and maximizes u such that $\mathcal{R}_1^l$, $\mathcal{R}_1^u$, $\mathcal{R}_2$ and $\mathcal{R}_3$ are satisfied. It will not return $[l, u]$ if $[l, u]$ is not maximal but this violates CR4. Since CR1 to CR4 are satisfied, the algorithm returns $[l, u]$ against our assumption.

We conclude that the algorithm computes D and the causal range $[l, u]$. This contradicts the assumption that the algorithm does not return $[l, u]$. □

9.4.2 Complexity Considerations

We elaborate on the complexity to compute the causal ranges for a TDT. The length n of a TDT is the number of its transitions. A TDT contains $n + 1$ different delays where each is in a causal range or not. For a TDT, there exists up to 2^{n+1} causal ranges.

Before we compute the causal ranges, we check for every delay in P1, whether it is causal or not. P1 uses existence and not-existence quantification. Linear-real arithmetic with quantifiers is in PSPACE [96]. Thus, P1 is in PSPACE and its complexity rises with the length of the TDT.

We find an existing causal range by P2. Since P2 is an optimization problem, we need to eliminate the quantifiers in Z3, which has a double exponential worst-case complexity in the number of quantifiers [40]. P2 solves two optimization problems where each can be solved in NP [96], since a linear optimization problem can be solved in polynomial time [83].

The algorithm executes P3 for every causal delay found by P1 that is not in a delay set D, which are at most n other delays. The number of constraints in P3 rises with the number of causal ranges existing for subsets of D, which is in the worst case 2^{n+1}. The satisfiability problem itself is in the worst case complexity class NP [96]. Thus, solving P3 has an exponential worst-case complexity.

Overall, the algorithm is dominated by exponentially many calls to the double exponential worst-case complexity of the quantifier elimination in P2. The algorithm has a double-exponential worst-case complexity. Despite this high worst-case complexity, we will see in the evaluation that exponentially many causal ranges do not exist in realistic models and that these are in practice computable within a reasonable amount of time.

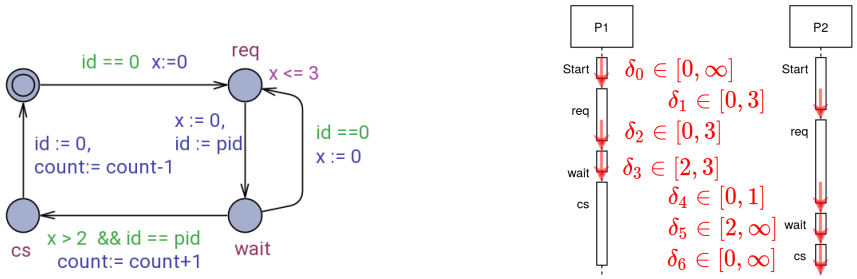
9.5 Evaluation

We implemented the Causal Range Algorithm in a tool that we call *Causal Timed Range Analyser* (CaTiRA) and evaluated the tool by computing causal ranges for several case studies taken from the literature.

Evaluation Methodology. CaTiRA is intended to be used at design time to support design space exploration based on causal information regarding the dynamic timing behavior. We foresee a usage of CaTiRA at an intermediate design stage when a considered preliminary model does not yet satisfy all required properties. The objective of the analysis is to point the designer to variations in the delay timings at certain locations in a TDT, which may motivate changes in, for instance, timing bounds in the model. Such preliminary design models are not available for experimentation. Therefore, it is necessary that we use published, correct models and revert them to a preliminary state by seeding syntactic code variations. We emphasize that the objective is not to locate these code variations, or even propose repairs to syntactic elements, but to illustrate to the designer what ranges of delay variations contribute to avoiding the observed property violation. The designer can then decide to perform syntactic changes to the model in order to constrain the timing of the system in such a way that property violations will be avoided.

Database Example [91]. For the TDT in Example 3, CaTiRA found the causal delays δ_2 and δ_4, and the causal ranges $2 \leq \delta_4 \leq 3$ and $3 \leq \delta_2 + \delta_4 \leq 5$.

Fischer's protocol [133]. The purpose of Fischer's protocol is to ensure mutual exclusion. The TA in Figure 9.2 is a process of the protocol with a unique id *pid* and requests for the critical section *cs*. The global variable *id* controls the access to the critical section and allows a process only to enter the critical location *cs* when id == pid. In order to request access to the critical section, a process checks the transition guard *id* == 0 to check that no other process currently requests access to the critical section, and subsequently enters the location *req*. Afterwards the process enters the location *wait* within 3 time units for the critical section and overwrites *id* with its unique *pid*. The process must wait 2 time units to ensure that if another



(a) TA (b) Sequence Diagram

Figure 9.2: Fischer's Protocol

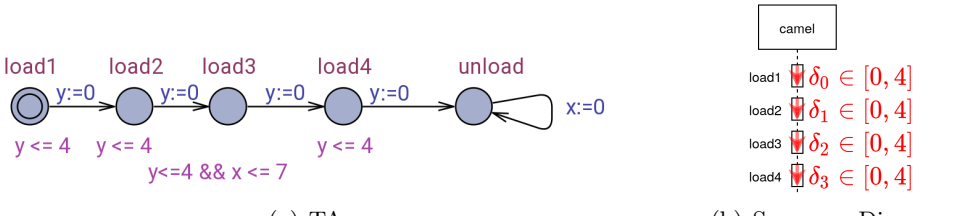
process also requests access to the critical section, that process has left the location *req*. In this model, the timing is not correct and it is possible that a process enters location *cs* and the other process is in location *req*. We reverted the model to a preliminary state by replacing the invariant $x \leq 2$ as in [133] by $x \leq 3$. We used UPPAAL to check mutual exclusion of a model with TAs *P1* and *P2*, and obtained a TDT depicted by the sequence diagram in Figure 9.2(b).

For this TDT, the causal range analysis by CaTiRA finds the causal delays δ_1 to δ_3, and the causal ranges $2 < \delta_1 \leq 3$, $0 \leq \delta_2 < 1$, $2 < \delta_3 \leq 3$, and $0 \leq \delta_1 + \delta_2 < 1$. Notice that the TDT of Fischer's protocol has no realization that satisfies the property. The causal ranges were computed because the Causal Range Algorithm also considers a suffix-blocking partial realization to satisfy the counterfactual argument (II).

The causal ranges with the causal delay variables δ_1, δ_2 and δ_3 are reasonable since during these time delays *P1* is in the location *req*, labeled with the seeded constraint $x \leq 3$. During the time delay δ_4, *P2* is in location *req*. However, δ_4 is not a causal delay variable since no different delay assignment exists for it that prevents the property violation. The interval of the causal ranges $2 < \delta_1 \leq 3$ and $2 < \delta_3 \leq 3$ is identical to the seeded faulty extension of the constraint. The choice of delay assignments that satisfies $0 \leq \delta_2 < 1$ and $0 \leq \delta_1 + \delta_2 < 1$ ensures that TA *P1* leaves location *req* early enough that the extension of the constraint comes into effect and the property violation will be reached. We see that the causal ranges actually express the choices of delay assignments that will lead to a property violation.

Camel Transporter (adapted from [33]) is a real-time model depicted in Figure 9.3(a) where in every location *load1* to *load4*, a worker loads a bag on a camel. The weight of a bag is between 1 and 4 units and is modeled by the time that the worker stays in a location. The camel will only arrive at the destination when the weight is not more than 7 units. The worker checks the payload of the camel with loading the third bag on the camel but is in a rush and does not check the payload after loading the fourth bag. A verification of the model with UPPAAL results in a TDT depicted in the Figure 9.3(b). We manually computed the possible assignments of the delay variables and added them in red to the sequence diagram. For this TDT, CaTiRA computes the causal delays δ_0 to δ_3, and the causal ranges $7 < \delta_0 + \delta_3 \leq 8$, $7 < \delta_1 + \delta_3 \leq 8$, $7 < \delta_2 + \delta_3 \leq 8$, $7 < \delta_0 + \delta_1 + \delta_3 \leq 11$, $7 < \delta_0 + \delta_2 + \delta_3 \leq 11$, $7 < \delta_1 + \delta_2 + \delta_3 \leq 11$, and $7 < \delta_0 + \delta_1 + \delta_2 + \delta_3 \leq 11$.

All causal ranges contain the delay variable δ_3 of the location *load4*. This is



(a) TA (b) Sequence Diagram

Figure 9.3: Camel Transporter

reasonable since an overload of the camel is checked in location *load3*. The load limit of 7 can only be exceeded in a combination of at least two delay assignments since the maximal delay assignment is 4 for every delay variable. Also, even two variable assignments can already result in an overload of the camel, every delay assignment has an impact on whether the camel is overloaded. This becomes obvious by the realization $\delta_0 = 2$, $\delta_1 = 2$, $\delta_3 = 2$, $\delta_4 = 2$ that is contained only by the causal range $7 < \delta_0 + \delta_1 + \delta_2 + \delta_3 \leq 11$. In this realization, every delay assignment contributes to the overload of the camel but no subset of the delay assignments can overload the camel. We see that the causal ranges express the dynamic timing behavior that leads to the property violation.

The Pacemaker Model [80] originally satisfies the property that the time delay between two ventricular heartbeats is not to high. We analyze this model since it is a realistic model and of a reasonable size. We artificially reverted the model to a preliminary state by replacing an invariant that controls the minimal heart period from 400 to 1,600 time units. This results in a property violation since the time delay between two ventricular heartbeats can be now too high. For the TDT illustrating the property violation, CaTiRA computes the causal delays δ_0 and δ_6, and the causal range $150 < \delta_6 < 350$.

The results can be interpreted as follows. After the time delay of δ_0, the first heartbeat happens and a timer starts to measure the time delay until the next ventricular heartbeat. For some realization of the TDT that violates the property, an increase of the value assignment of δ_0 can prevent the property violation, which means that, δ_0 is a causal delay variable. However, no causal range with δ_0 exists since each of these realizations is already contained by the causal range $150.0 < \delta_6 < 350.0$. Only during the time delay δ_6, the TDT is in the location in which a constraint was altered when manipulating the model. The modification of the constraint corresponded to an increase of 1200 time units of a bound. However, the assignments in this location that cause a property violation are in the range from $[150, 350]$. Thus, only the increase of the constraint bound by the first 200 time units has an impact on the possible execution. We see that the causal range shows the erroneous timing behavior of the TDT.

Quantitative Results. The quantitative results of every model are represented in Table 9.1. For every model, we indicate the time T_{UP} that UPPAAL needed to compute a TDT for the model and the length Ln of the TDT. For a given TDT, CaTiRA computes a number $\#D$ of causal delay variables and a number $\#R$ of causal ranges. The computation of the causal ranges takes in total a time T in seconds and consumes at most an amount M of memory in megabytes. Z3 solves constraint systems with at most a count $\#Cn$ of clauses and at most a number $\#Vr$ of variables. Z3 needs at most the time T_{Z3} in seconds to solve a constraint system with a maximal memory usage of M_{Z3} in megabytes.

All experiments were performed with the SMT-solver Z3 (Version 4.8.3) on a computer with an i7-6700K CPU (4.00GHz), 60GB of RAM and a Linux operating system. For the considered models, we found a total of 11 causal delay variables and 14 causal ranges. The highest computation effort for a causal range computation can

Model	T_{UP}	Ln	$\#D$	$\#R$	T	M	$\#Cn$	$\#Vr$	T_{Z3}	M_{Z3}
database [91]	0.010	4	2	2	0.420	13.2	66	19	0.020	6.7
Fischer's protocol [133]	0.010	6	3	4	0.626	26.5	78	23	0.040	6.9
Camel Transporter [33]	0.008	3	4	7	4.724	35.9	138	105	0.350	7.1
Pacemaker [80]	0.015	7	2	1	0.587	33.5	226	114	0.080	7.3

Table 9.1: Quantitative Experimental Results of Causal Range Analysis.

be observed in the camel transporter TDT with 35.9 MB memory consumption and 4.724s computation time. In line with this, Z3 has the highest computation effort in time (0.350s) with this model. The intrinsic complexity of this TDT seems to be high since with 138 clauses, it has fewer clauses than the TDT of the Pacemaker model with 226 clauses.

The most complex model is the Pacemaker model since it takes the most time (0.015) for UPPAAL to compute the TDT. Also, its TDT is the longest with 7 transitions. With 226 the encoding of the analysis has the most clauses and with 114 the most variables, even though. The computation effort of the causal ranges is moderate with 0.587s and 33.5 MB. In conclusion, the analysis results show that the causal range analysis requires only a reasonable computation effort, in spite of its theoretical worst-case complexity.

9.6 Conclusion

We have presented the Causal Range Algorithm and its implementation in the tool CaTiRA. Based on a counterfactual causality argument, the Causal Range Algorithm performs an analysis to determine dynamic causes for timed safety property violations in the timing behavior of a timed system, as documented by TDTs. Using various case studies, we have shown that the analysis is both efficient and effective. In particular, our work shows that using interpretations of counterfactual causal reasoning can lead to precise and intuitive explanations for dynamic timing behaviors.

Another possibility to explain a property violation is to analyze the syntactic features of a TDT, as we will see in the next chapter.

Chapter 10

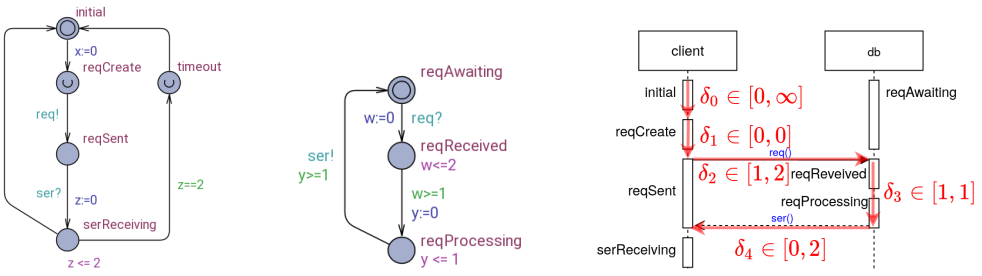
Repairs for Timed Automata

Contents

The content of this part is based on the publications [88, 91, 92].

10.1 Introduction

We present an automated method that computes static causes in form of proposals for possible repairs of an NTA that avoid the violation of a timed safety property. Consider the TDT depicted as a time annotated sequence diagram in Figure 10.1(c). This scenario is similar to the scenario in Example 3 and describes a simple message exchange where the process `client` sends a message `req` to the process `db` which, after some processing steps, returns a message `ser` to `client`. The scenarios differ only in the timing of message `ser`, which has to be received in Figure 10.1(c) in 2 time units, and not in 3 time units, as in Figure 8.1(c). Assume a requirement on the system stating that the time from sending `req` to receiving `ser` is not to be more than 4 time units. Further assume that the timing interval annotations on



(a) TA `client` (b) TA `db` (c) TDT illustrated using a time-interval annotated sequence diagram

Figure 10.1: Modified Database Example

the sequence diagram represent the minimum and maximum time for the message transmission and processing steps that the NTA, from which the diagram has been derived, permits. It is then easy to see that it is possible to execute the system in such a way that this property is violated.

Various changes to the underlying NTA model depicted in Figure 10.1 may avoid this property violation. We present analyses that can suggest a whole range of repairs in addition to clock bound variation, such as modifying clock bounds, comparison operators in constraints, clock references, clock resets, and location urgency. Examples of repairs computed by our analyses for the model in Figure 10.1 are:

- Reducing the maximum time it takes to transmit `ser` message to be at most 1 time unit by exchanging $z <= 2$ with $z <= 1$.
- Exchanging the comparison operator in the constraint $w \geq 1$ to $w < 1$ ensures that the time to send a request is below 1 time unit.
- An exchange of clock z in $z \leq 2$ with clock y restricts the time of processing and receiving the response to at most 2 time units.
- To reset the clock y on the previous transition instead ensures that the time for sending and processing the request is below 1 time unit.
- Making the location *serReceiving* urgent reduces the time to receive a response to 0.

Proposing such changes to the model may either serve to correct mistakes made during the editing of the model, or point to necessary changes in the dimensioning of its time resources, thus contributing to improved design space exploration.

The repair method described in this chapter relies on an encoding of a TDT as a constraint system as described in Chapter 8. This encoding provides symbolic abstract semantics for the TDT by constraining the sojourn time of the NTA in the locations visited along the trace. The constraint system is then augmented by auxiliary model variation variables which represent syntactic changes to the NTA model, for instance, the variation of a location invariant condition or a transition guard. We claim that the constraint system modified in this way implies the non-reachability of a violation. At the same time, we assert that the model variation variables have a value that implies that no change of the NTA model will occur, for instance, by setting a clock bound variation variable to 0. This renders the resulting constraint system unsatisfiable.

In order to compute a repair, we derive a partial MaxSMT instance by turning the constraints that disable any repair into soft constraints. We solve this MaxSMT instance using the SMT solver Z3 [42]. If the MaxSMT instance admits a solution, the resulting model provides values of the model variation variables. These values indicate a repair of the NTA model which entails that along the sequence of locations represented by the TDT, the property violation will no longer be reachable.

In a next step it is necessary to check whether the computed repair is an admissible repair in the context of the full NTA. This is important since the repair

was computed locally with respect to only a single given TDT. Thus, it is necessary to define a notion of *admissibility* that is reasonable and helpful in this setting. To this end, we propose the notion of *functional equivalence* which states that as a result of the computed repair, neither erstwhile existing functional behavior will be eliminated, nor will new functional behavior be added. Functional behavior in this sense is represented by the languages accepted by the untimed automata of the unrepaired and the repaired NTAs. Functional equivalence is then defined as equivalence of the languages accepted by these automata. We propose a zone-based automaton construction for implementing the functional equivalence test that is efficient in practice.

We have implemented our proposed method in a proof-of-concept tool called TarTar. Our evaluation of TarTar is based on several non-trivial NTA models taken from the literature, including the frequently considered pacemaker model [80]. For each model, we automatically generate mutants by injecting syntactic modifications which we then model check using UPPAAL and repair using TarTar. The evaluation shows that our technique is able to compute an admissible repair for 69% to 88% of the detected faults.

Structure of the Chapter. We present the repair analyses in Section 10.2 and the admissibility analysis in Section 10.3. The implementation in the tool TarTar is presented in Section 10.4. In Section 10.5, we report on experimental evaluation and case studies. A conclusion is provided in Section 10.6.

10.2 Repair Computations

We propose a repair technique that analyzes the responsibility of syntactic elements occurring in a single TDT for causing a violation of a specification Π. We first present the formalization of different syntactic modifications of a TDT that can potentially remedy causes for property violations and then present the algorithm to compute a set of such syntactic modifications that are possible repairs. The question of admissibility of the computed repairs will be addressed in Section 10.3. Throughout this section, we assume that S is a TDT for a TA T and Π.

10.2.1 Formal Modification of Syntactic Elements

We introduce *variation variables* v that represent possible correction values of syntactic elements of the TA for which S is produced. They represent possible repairs of the TA model. For instance, to modify clock bounds for a transition in a TA, a fresh variation variable is added to every clock bound occurring in location invariants and transition guards involved in this transition. The values of the variation variables are computed so that none of the realizations of S in the modified automaton leads to a violation of Π. This is done by defining a new constraint system that captures the conditions on the variable v under which the violation of Π will not occur in the corresponding trace of the modified automaton. Using this constraint system, we then define a MaxSMT problem so that its solution minimizes the number of changes to T that are needed to achieve the repair.

Clock Bound Variation. Recall that the clock bounds occurring in location invariants and in transition guards are represented by the $\mathcal{I}$ and $\mathcal{G}$ sets defined for the TDT S in Definition 37. We then introduce bound variation variables v_i^{bv} describing the possible static variation in the TA code for a clock bound β with an index i in the set $ibounds \cup gbounds$, and modify the TDT constraint system accordingly. A variation of the bounds only affects the location invariant constraints $\mathcal{I}$ and the transition guard constraints $\mathcal{G}$. We thus define an appropriate invariant variation constraint $\mathcal{I}^{bv}$ and guard variation constraint $\mathcal{G}^{bv}$ that capture the clock bound modifications:

$$\mathcal{I}^{bv} \equiv \bigwedge_{(c\sim\beta)\in ibounds} c \sim (\beta + v_i^{bv}) \wedge c + \delta_j \sim (\beta + v_i^{bv})$$
$$\mathcal{G}^{bv} \equiv \bigwedge_{(c\sim\beta)\in gbounds} c + \delta_j \sim (\beta + v_i^{bv})$$

We also need constraints ensuring that the modified clock bounds remain positive:

$$\mathcal{Z}^{bv} \equiv \bigwedge_{(c\sim\beta)\in ibounds \cup gbounds} \beta + v_i^{bv} \geq 0$$

Putting all of this together, we obtain the *bound variation TDT constraint system*

$$\mathcal{T}^{bv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{bv} \wedge \mathcal{G}^{bv} \wedge \mathcal{Z}^{bv} \wedge \mathcal{U} \wedge \mathcal{L}$$

which captures all realizations of S in TAs T^{bv} that are obtained from T by modifying the clock bounds by some syntactically consistent variations v_i^{bv}.

As an example, consider the bound variation for the guard $y \geq 1$ of transition Θ_3 in Figures 10.1(b). The modified guard constraint, a conjunct in $\mathcal{G}^{bv}$, is $y_3 + \delta_3 \geq 1 + v_i^{bv}$. The corresponding non-negativity constraint in $\mathcal{Z}^{bv}$ is $1 + v_i^{bv} \geq 0$.

Operator Variation. This type of variation is motivated by the assumption that a wrong comparison operator in a location invariant or transition guard may cause a property violation. We assume for the repair encoding that the operators $\sim$ are indexed according to their order in the sequence $\langle <, \leq, =, \geq, > \rangle$. The possible repairs are encoded by a fresh variation variable v_i^{ov} where the value of v_i^{ov} is the index of the corresponding comparison operator. For instance, the use of the $<$ comparison operator as a repair is indicated by setting $v_i^{ov} = 1$. $v_i^{ov} = 0$ indicates to use the original operator.

We define operator variation constraints $\mathcal{I}^{ov}$ and $\mathcal{G}^{ov}$ with the help of an n-ary exclusive or operation $\bigoplus_{i=0\ldots n} f_i$ which is satisfied iff exactly one of the formulas f_i is true:

$$\mathcal{I}^{ov} \equiv \bigwedge_{(c\sim\beta)\in ibounds} \bigoplus_{0\leq k\leq 5} c \sim_k \beta \wedge c + \delta_j \sim_k \beta \wedge v_i^{ov} = k.$$
$$\mathcal{G}^{ov} \equiv \bigwedge_{(c\sim\beta)\in gbounds} \bigoplus_{0\leq k\leq 5} c + \delta_j \sim_k \beta \wedge v_i^{ov} = k.$$

Now, we construct an operator variation TDT constraint system $\mathcal{T}^{ov} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{ov} \wedge \mathcal{G}^{ov} \wedge \mathcal{U} \wedge \mathcal{L}$.

As an example for the operator variation encoding, consider the guard $w >= 1$ of transition θ_2 in Figures 10.1(b). The $\mathcal{G}^{ov}$ contains the constraint $w_2 >= 1 \wedge (v_i^{ov} = 0) \oplus ... \oplus w_2 > 1 \wedge (v_i^{ov} = 5)$.

Clock Reference Variation. This variation aims at repairing property violations resulting from errors that can be traced back to the unintended use of a wrong clock variable. We uniquely identify every clock in the repair encoding by an index k. We introduce a fresh variation variable v_i^{cv} for every constraint with a clock c. Its assigned value k indicates that in the constraint, the clock c_k is to be used instead of c. For example, if $y \leq 2$ is a repaired constraint and clock y has index $k = 1$, then $v_i^{cv} = 1$ holds.

We define the appropriate clock variation constraints $\mathcal{I}^{cv}$ and $\mathcal{G}^{cv}$:

$$\mathcal{I}^{cv} \equiv \bigwedge_{(c\sim\beta)\in ibounds} \bigoplus_{0\leq k\leq|C|} (c_k \sim \beta) \wedge (c_k + \delta_j \sim \beta) \wedge (v_i^{cv} = k)$$
$$\mathcal{G}^{cv} \equiv \bigwedge_{(c\sim\beta)\in gbounds} \bigoplus_{0\leq k\leq|C|} (c_k + \delta_j \sim \beta) \wedge (v_i^{cv} = k)$$

From this we obtain the clock reference variation TDT constraint system $\mathcal{T}^{cv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I}^{cv} \wedge \mathcal{G}^{cv} \wedge \mathcal{U} \wedge \mathcal{L}$.

An example for the clock reference repair encoding is given by the guard $y \geq 1$ of transition θ_3 in Figures 10.1(b). $\mathcal{G}^{cv}$ contains the constraint $(y_3 + \delta_3 \geq 1) \wedge (v_i^{cv} = 0) \oplus ... \oplus (z_3 + \delta_3 \geq 1) \wedge (v_i^{cv} = 4)$.

Reset Clock Variation. This variation aims at repairing a property violation by adding or removing clock resets. We introduce a variation variable $v_{c,\theta}^{rv}$ for the transition θ leaving location l_j and every clock c in the TDT. The reset status in the extended constraint system is inverted when $v_{c,\theta}^{rv} \neq 0$: if c was not reset before, it will now be reset, and vice versa. This is encoded by the clock reset variation constraints $\mathcal{R}^{rv}$ and $\mathcal{D}^{rv}$:

$$\mathcal{R}^{rv} \equiv \bigwedge_{c\in reset(\theta_j)} c_{j+1} = \begin{cases} 0, & \text{if } v_{c,\theta}^{rv} = 0 \\ c + \delta_j, & \text{otherwise} \end{cases} .$$
$$\mathcal{D}^{rv} \equiv \bigwedge_{c\notin reset(\theta_j)} c_{j+1} = \begin{cases} c + \delta_j, & \text{if } v_{c,\theta}^{rv} = 0 \\ 0, & \text{otherwise} \end{cases} .$$

As a result we obtain the reset clock variation TDT constraint system $\mathcal{T}^{rv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R}^{rv} \wedge \mathcal{D}^{rv} \wedge \mathcal{I} \wedge \mathcal{G} \wedge \mathcal{U} \wedge \mathcal{L}$.

To illustrate the clock reset repair encoding, consider the clock reset $y := 0$ on transition θ_2 in Figures 10.1(b). $\mathcal{R}^{rv}$ contains the constraint $((y_3 = 0) \wedge (v_{y,\theta_2}^{rv} = 0)) \vee ((y_3 = y_2 + \delta_2) \wedge (v_{y,\theta_2}^{rv} \neq 0))$.

Urgent Location Variation. Here we aim at repairing cases where a faulty usage of urgent locations causes a property violation. Urgency of a location is modeled in the TDT constraint system by setting the location delay δ_j to 0. We define a fresh

variation variable v_i^{uv} for a location l_j. For $v_i^{uv} \neq 0$, the urgency for a location l_j is inverted. We encode this idea using the following urgency variation constraint $\mathcal{U}^{uv}$:

$$\mathcal{U}^{uv} \equiv \bigwedge_{l_j \in urgent} (v_i^{uv} = 0 \implies \delta_j = 0) \quad \wedge \bigwedge_{l_j \notin urgent} (v_i^{uv} \neq 0 \implies \delta_j = 0).$$

We construct the urgent location variation TDT constraint system $\mathcal{T}^{uv} \equiv \mathcal{C}_0 \wedge \mathcal{A} \wedge \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{I} \wedge \mathcal{G} \wedge \mathcal{U}^{uv} \wedge \mathcal{L}$.

As an example for the urgent location repair encoding, consider the location `serReceiving` reached by θ_3 in Figures 10.1(a). $\mathcal{U}^{uv}$ contains the constraint $(v_i^{uv} \neq 0) \rightarrow (\delta_4 = 0)$.

10.2.2 Repair by Variation Analysis.

The objective of the variation analysis is to provide hints to the system designer regarding which minimal syntactic changes to the considered model might prevent the violation of property Π. Minimality is interpreted in the number of syntactic modifications that need to be performed in the considered TA model.

For the analysis, we first choose one of the variation TDTCS described before and denote it with $\mathcal{T}^*$. We implement an analysis by using $\mathcal{T}^*$ to derive an instance of the partial MaxSMT problem whose solutions yield candidate repairs for the timed automaton T. The partial MaxSMT problem takes as input a finite set of assertion formulas belonging to a fixed first-order theory. These assertions are partitioned into *hard* and *soft* assertions. The hard assertions $\mathcal{F}_H^*$ are assumed to hold and the goal is to find a maximal subset $\mathcal{F}' \subseteq \mathcal{F}_S^*$ of the soft assertions such that $\mathcal{F}' \cup \mathcal{F}_H^*$ is satisfiable in the given theory.

The property Π is encoded as a constraint Φ by Definition 37. For the purpose of our analysis, the hard assertions are formed by the conjunction

$$\mathcal{F}_H^* \equiv (\exists \delta_j, c_j . \mathcal{T}^*) \wedge (\forall \delta_j, c_j . \mathcal{T}^* \Rightarrow \Phi).$$

We refer by v_i^* to a variation variable in $\mathcal{F}_H^*$. Note that the free variables of $\mathcal{F}_H^*$ are exactly the variation variables v_i^*. Given a satisfying assignment ι for $\mathcal{F}_H^*$, let T_ι be the timed automaton obtained from T by modifying a clock bound according to the variation value $\iota(v_i^*)$ and let S_ι be the TDT corresponding to S in T_ι. Then $\mathcal{F}_H^*$ guarantees that

1. S_ι is feasible, and
2. S_ι has no realization that witnesses a violation of Π in T_ι.

We refer to such an assignment ι as a *local clock repair* for T and S. To obtain a minimal local clock repair, we use the soft assertions given by the conjunction

$$\mathcal{F}_S^* \equiv \bigwedge_{v_i^*} v_i^* = 0.$$

Clearly, $\mathcal{F}_H^* \wedge \mathcal{F}_S^*$ is unsatisfiable because $\mathcal{T}^* \wedge \mathcal{F}_S^*$ is equisatisfiable with $\mathcal{T}$, and $\mathcal{T} \wedge \neg\Phi$ is satisfiable by assumption. However, if there exists at least one local clock

repair for T and S, then $\mathcal{F}_H^\star$ alone is satisfiable. In this case, the MaxSMT instance $\mathcal{F}_H^\star \cup \mathcal{F}_S^\star$ has at least one solution. Every satisfying assignment of such a solution corresponds to a local repair that minimizes the number of syntactic modifications that need to be performed on S, and hence on the considered TA. Note that hard and soft assertions remain within a decidable logic. Using an SMT solver such as Z3, we enumerate every optimal solution for the partial MaxSMT instance and obtain a minimal local clock bound repair from each of them. For an enumeration, we prevent a found optimal solution by hard asserting the in the solution modified variables to value 0.

Example 4. *We have applied the bound variation repair analysis to the TDT from Figure 10.1(c). The following repairs exist:*

1. $v^{bv}_{z,l_4} = -1$. *This corresponds to a variation of the location invariant regarding clock* **z** *in location* 4 *of the TDT, corresponding to location* **client.serReceiving**, *to read* $z \leq 1$ *instead of* $z \leq 2$. *This indicates that the violation of the bound on the total duration of the transaction, as indicated by a return to the* **serReceiving** *location and a value greater than 4 for clock* **x**, *can be avoided by ensuring that the time taken for transmitting the* **ser** *message to the* **client** *is constrained to take less or equal* 1 *time unit.*

2. *A further computed repair is* $v^{bv}_{w,l_2} = -1$. *Interpreting this variation in the context of Figure 10.1(b) means that location* **db.reqReceived** *will be left when the clock* **w** *has value* 1. *In other words, the transmission of the message* **req** *to the* **db** *takes exactly one time unit, not between* 1 *and* 2 *time units as in the unrepaired model.*

3. *Another computed repair is* $v^{bv}_{y,l_3} = -1$ *and* $v^{bv}_{y,\theta_3} = -1$, *which reduce the time delay in location* **db.reqProcessing** *by 1 time unit.*

4. *No further minimal repair exists.*

Examples of repairs according to the other analyses for the TDT in Figure 10.1(c) include the following. The operator repair analysis returns, among others, a repair $v^{ov}_{w,\theta_2} = 1$, which suggests to exchange the operator in the transition guard $w \geq 1$ by $<$ and to reduce the time for receiving the message `req` to be less than one time unit. The reference clock analysis returns a repair $v^{cv}_{z,l_4} = y$. The repair exchanges the clock z in the clock constraint $z \leq 2$ by y, which implies that the model has to receive the message `ser` in less than one time unit, since the guard $y \geq 1$ is satisfied when entering location `serReceiving`. The reset clock analysis returns a repair $v^{rv}_{x,\theta_2} = true$, which indicates to reset clock x during the transition leading into location `reqProcessing`. The urgent location analysis returns a repair $v^{uv}_{serReceiving} = true$ which suggests to turn `serReceiving` into an urgent location. This repair reduces the time for receiving message `ser` to zero.

Complexity Considerations. The tool Z3 [112] computes a repair for partial MaxSMT instances without quantifiers. The repair analysis first executes quantifier elimination on $\mathcal{F}_H^\star$, which has a double exponential worst-case complexity in

the number of quantifier variables [40]. The number of quantified variables in $\mathcal{F}_H^\star$ rises linearly with the length of the TDT. Afterwards, every repair analysis solves a MaxSMT problem for quantifier-free linear real arithmetic constraints. The complexity of the repair computation is determined by the number of time constraints in the TDT which increases with the length of the TDT. The complexity of quantifier-free linear real arithmetic is polynomial [83, 96]. The MaxSMT problem can be reduced in polynomial time to a MaxSAT problem, and a MaxSAT problem is an optimization problem in the complexity class NP [95]. We conclude that the quantifier elimination is the most complex computation in the analysis, which means that the overall complexity is double exponential in the length of the TDT in the worst case.

10.3 Admissibility of Repair

The synthesized repairs that lead to a TA T_ι change the original TA T in fundamental ways, both syntactically and semantically. This brings up the question whether the synthesized repairs are admissible. In fact, one of the key questions is what notion of admissibility is meaningful in this context.

Admissibility Criteria. From a *syntactic* point of view, the repair obtained from a satisfying assignment ι of the MaxSMT instance ensures that T_ι is a syntactically valid TA model, for instance, by placing non-negativity constraints on repaired clock bounds. In case repairs alter right hand sides of clock constraints to rational numbers, this can easily be transformed to integers by normalizing all clock constraints in the TA.

From a *semantics* perspective, the impact of the repairs is more profound. Since the repairs affect time bounds in location invariants and transition guards, as well as clock resets, the behavior of T_ι may be fundamentally different from the behavior of T.

- The computed repair for one property Π may render another property Π' violated. To check admissibility of the synthesized repair with respect to the set of all properties $\widehat{\Pi}$ in the system specification, a full re-checking of $\widehat{\Pi}$ is necessary.

- A repair may have introduced zenoness and timelock [14] into T_ι. As discussed in [14], there exist both an over-approximating static test for zenoness as well as a model checking based precise test for timelocks that can be used to verify whether the repair is admissible in this regard.

- Due to changes in the possible assignment of time values to clocks, reachable locations in the TA T may become unreachable in T_ι, and vice versa. On the one hand, this means that some functionalities of the system may no longer be provided since part of the actions in T will no longer be executable in T_ι, and vice versa. Further, a reduction in the set of reachable locations in T_ι compared to T may mean that certain locations with property violations in T are no longer reachable in T_ι, which implies that certain property violations

are masked by a repair instead of being fixed. On the other hand, the repair leading to locations becoming reachable in T_ι that were unreachable in T may have the effect that previously unobserved property violations become visible and that T_ι possesses functionality that T does not have, which may or may not be desirable.

It should be pointed out that we assess admissibility of a repair leading to T_ι with respect to a given TA model T, and not with respect to a correct TA model T^* satisfying Π. We define an admissibility test based on functional equivalence.

Functional Equivalence. While various variants of semantics admissibility may be considered, we are focusing on a notion of admissibility that ensures that a repair does not unduly change the functional behavior of the modeled system while adhering to the timing constraints of the repaired system. We refer to this as *functional equivalence*. The functional capabilities of a timed system manifest themselves in the sets of action or transition traces that the system can execute. For TAs T and T_ι this means that we need to consider the languages over the action or transition alphabets that these TAs define. Considering the timed languages of T and T_ι, we can state that $\mathcal{L}_T(T) \neq \mathcal{L}_T(T_\iota)$ since the repair forces at least one timed trace to be purged from $\mathcal{L}_T(T)$. This means that equivalence of the timed languages cannot be an admissibility criterion ensuring functional equivalence.

At the other end of the spectrum we may relate the de-timed languages of T and T_ι. The *de-time* operator $\alpha(T)$ is defined such that it omits all timing constraints and resets from any TA T. Requiring $\mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$ is tempting since it states that when eliminating all timing related features from T and from the repaired T_ι, the resulting action languages will be identical. However, this admissibility criterion would be flawed, since the repair in T_ι may imply that unreachable locations in T will be reachable in T_ι, and vice versa. This may have an impact on the untimed languages, and even though $\mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$, it may be that $\mathcal{L}_\mu(T) \neq \mathcal{L}_\mu(T_\iota)$. To illustrate this point, consider the running example in Fig. 10.1(a) and assume the invariant in location `client.serReceiving` to be modified from $z \leq 2$ to $z \leq 1$ in the repaired TA T_ι. Applying the de-time operator to T_ι implies that the location `client.timeout`, which is unreachable in T_ι, becomes reachable in the de-timed model. Since `client.timeout` is reachable in T, the TAs T and T_ι are not functionally equivalent, even though their de-timed languages are identical. Notice that for the untimed languages it holds that $\mathcal{L}_\mu(T) \neq \mathcal{L}_\mu(T_\iota)$, since no timed trace in $\mathcal{L}_T(T_\iota)$ reaches location `timeout`, even though such a timed trace exists in $\mathcal{L}_T(T)$. In particular, $\mathcal{L}_\mu(T)$ contains the untimed trace $\Theta_0\Theta_1\Theta_2\Theta_3\Theta_4$, where Θ_4 is the transition towards the location `client.timeout`, which is missing in $\mathcal{L}_\mu(T_\iota)$. As a consequence, we resort to considering the untimed languages of T and T_ι when assessing functional equivalence and require $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$. It is easy to see that $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota) \Rightarrow \mathcal{L}(\alpha(T)) = \mathcal{L}(\alpha(T_\iota))$. In conclusion, the equivalence of the untimed languages ensures functional equivalence.

Admissibility Test. Designing an algorithmic admissibility test for functional equivalence is challenging due to the computational complexity of determining the

equivalence of the untimed languages $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. While language equivalence is decidable for languages defined by NBAs, it is undecidable for timed languages [7]. For untimed languages, however, this problem is again decidable [7]. The algorithmic implementation of the test for functional equivalence that we propose proceeds in two steps.

- First, the untimed languages $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$ are constructed. This requires an untime transformation of T and T_ι, yielding NBAs representing $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. While the standard untime transformation for TAs [7] relies on a region construction, we propose a transformation that relies on a zone construction [68]. This will provide a more succinct representation of the resulting untimed languages and, hence, a more efficient equivalence test.
- Second, it needs to be determined whether $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$. As we shall see, the obtained NBAs are closed. Hence, we can reduce the equivalence problem for these ω-regular languages to checking equivalence of the regular languages obtained by taking the finite prefixes of the traces in $\mathcal{L}_\mu(T)$ and $\mathcal{L}_\mu(T_\iota)$. This allows us to interpret the NBAs obtained in the first step as NFAs, for which the language equivalence check is a standard construction [70].

Automata for Untimed Languages. The construction of an automaton representing an untimed language, here referred to as an *untime construction*, has so far been proposed based on a region abstraction [7]. The region abstraction is known to be relatively inefficient since the number of regions is, among other things, exponential in the number of clocks [14]. We therefore propose an untime construction based on the construction of a zone automaton [68], which in the worst case is of the same complexity as the region automaton, but is more succinct in practice [27].

Definition 40 (Untimed Nondeterministic Büchi Automaton)**.** *Assume a TA T and the corresponding zone automaton $[\![T]\!]_Z = (S_Z, s_Z^0, \Sigma_Z, \Theta_Z)$. We define the* untimed Nondeterministic Büchi automaton *as the closed NBA $B_T = (S, \Sigma, \to, S_0, F)$ obtained from $[\![T]\!]_Z$ such that $S = S_Z$, $\Sigma = \Sigma_Z \setminus \{\delta\}$, $S_0 = \{s_Z^0\}$ and $F = S$. For every transition in Θ_Z with a label $a \in \Sigma$, we add a transition to $\to$ created by the rule $\frac{(l,z)\overset{\delta}{\rightsquigarrow}(l,z^\uparrow)\overset{a}{\rightsquigarrow}(l',z')}{(l,z)\overset{a}{\to}(l',z')}$ with $z^\uparrow = \{v + \delta \in I(l) \mid v \in z, \delta \in \mathbb{R}_{\geq 0}\}$. In addition, we add self-transitions $(l,z) \overset{\tau}{\to} (l,z)$ to every state $(l,z) \in S_B$.*

The following observations justify this definition:

- A timed trace of T may remain forever in the same location after a finite number of action transitions. In order to enable B_T to accept this trace, we add a self-transition labeled with τ to $\to$ for each state $s \in S$ in B_T, and later define s as accepting. These τ-self-transitions extend every finite timed trace t leading to a state in S_τ to an infinite trace $t.\tau^\omega$.
- The construction of the acceptance set F is more intricate. Convergent traces are often excluded from consideration in real-time model checking [14]. As

a consequence, in the untime construction proposed in [7], only a subset of the states in S may be included in F. A repair may render a subgraph of the location graph of T that is only reachable by divergent traces, into a subgraph in T_ι that is only reachable by convergent traces. However, excluding convergent traces is only meaningful when considering unbounded liveness properties, but not when analyzing timed safety properties, which in effect are safety properties. As argued in [27], unbounded liveness properties appear to be less important than timed safety properties in timed systems. This is due to the observation that divergent traces reflect unrealistic behavior in the limit, but finite prefixes of infinite divergent traces, which only need to be considered for timed safety properties, correspond to realistic behavior. This observation is also reflected in the way in which, e.g., UPPAAL treats reachability by convergent traces. In conclusion, this argument justifies our choice to define the zone automaton in the untime construction as a closed BA, i.e., $F = S$.

We will now prove Theorem 16 and Theorem 17 to show that our admissibility test is correct. Theorem 16 states that the zone based untimed NBA construction actually preserves the untimed languages. In particular, we show that for a given TA T, $\mathcal{L}(B_T) = \mathcal{L}_\mu(T)$. For the proof, we use *Untimed Bisimulation* in order to replace the concrete time delay in T by the empty word ϵ, and to establish the desired language equality.

Definition 41 (Untimed Bisimulation (adapted from [27]))**.** *For a TTS $H = (S, s_0, \Sigma, \rightarrow)$, let $\rightarrow_\epsilon$ be the relation obtained from $\rightarrow$ by replacing all delay transitions (s_1, δ, s_2) by (s_1, ϵ, s_2). Then, a TTS $H = (S, s_0, \Sigma, \rightarrow)$ is said to* untimed simulate *another TTS $H' = (S', s'_0, \Sigma, \rightarrow')$ if there exists a relation $R \subseteq S \times S'$ such that (1) $(s_0, s'_0) \in R$ and (2) for all $(s_1, s'_1) \in R$, $s_2 \in S$, and $a \in \Sigma \cup \{\epsilon\}$ it holds that if $s_1 \xrightarrow{a} s_2 \in \rightarrow_\epsilon$, then there exists s'_2 with $s'_1 \xrightarrow{a} s'_2 \in \rightarrow'_\epsilon$ and $(s_2, s'_2) \in R$. H and H' are* untimed bisimilar *if in addition, H' untimed simulates H for R.*

The proof of Theorem 16 is based on the insight from Lemma 9 that two untimed bisimilar TTSs have equivalent untimed languages.

Lemma 9. *Given two TTS H_1 and H_2, if H_1 untimed simulates H_2, then $\mathcal{L}_\mu(H_1) \subseteq \mathcal{L}_\mu(H_2)$.*

Proof. Assume that TTS $H_1 = (S, s_0, \Sigma, \rightarrow)$ untimed simulates a TTS $H_2 = (S', s'_0, \Sigma, \rightarrow')$ with a simulation relation R. For any word v in $\mathcal{L}_\mu(H_1)$, we know that a timed trace $\xi = (t_1, a_1) \ldots$ exists in $\mathcal{L}(H_1)$. We show that v is also in $\mathcal{L}_\mu(H_2)$ by inductively constructing a timed trace $\xi' = (t'_1, a'_1) \ldots$ in $\mathcal{L}(H_2)$ with $\mu(\xi) = \mu(\xi')$. The induction iterates over the timed actions (t_i, a_i) in ξ.

Base Case: Initially, no timed action is taken in ξ. The relation (s_0, s'_0) is in R. For the induction step, we assume helper variables $t_0 = 0$ and $t'_0 = 0$.

Induction Step: For a timed action (t_{i+1}, a_{i+1}) in ξ and (s_i, s'_i) in R, we now construct a timed action (t'_{i+1}, a'_{i+1}) in ξ'.

For (t_{i+1}, a_{i+1}), a state s_* exists in H_1 such that a delay transition (s_i, t, s_*) with $t = t_{i+1} - t_i$ and an action transition (s_*, a_{i+1}, s_{i+1}) are in $\rightarrow$. The untimed

bisimulation replaces delay transitions by ϵ, and defines $\rightarrow_\epsilon$ and $\rightarrow'_\epsilon$. Since (s_i, s'_i) is in R and $(s_i, \epsilon, s_*) \in \rightarrow_\epsilon$, a state s'_* has to exist with $(s'_i, \epsilon, s'_*) \in \rightarrow'_\epsilon$ and $(s_*, s'_*) \in R$. (s'_i, ϵ, s'_*) exists only since a delay transition (s'_i, t', s'_*) exists in $\rightarrow'$. We construct $t'_{i+1} = t'_i + t'$. Because (s_*, a_{i+1}, s_{i+1}) is in $\rightarrow_\epsilon$ and $(s_*, s'_*) \in R$, a state s'_{i+1} in H_2 has to exist such that $(s_{i+1}, s'_{i+1}) \in R$ and $(s'_*, a_{i+1}, s'_{i+1}) \in \rightarrow'_\epsilon$. We conclude $a'_{i+1} = a_{i+1}$. Hence, we constructed the timed action (t'_{i+1}, a'_{i+1}) of ξ' and $(s_{i+1}, s'_{i+1}) \in R$ holds.

In conclusion, we can construct for every timed word ξ in $\mathcal{L}(H_1)$, a timed word ξ' in $\mathcal{L}(H_2)$, such that $\mu(\xi) = \mu(\xi')$. □

Additionally, the proof of Theorem 16 relies on the assumption that the timed language of the zone automaton $[\![T]\!]_Z$ is identical to the timed language of T, which we state in the following lemma.

Lemma 10 (Zone Language Equivalence)**.** *Let $[\![T]\!]_Z$ be a zone automaton derived from a TA T, then $\mathcal{L}_T([\![T]\!]_Z) = \mathcal{L}_T(T)$.*

Proof. The proof of this lemma can be derived from the proofs of Theorem 1 and Theorem 2 given in the full version of [141], which together prove that the reachable states in the transition system of T are also reachable in $[\![T]\!]_Z$, and vice versa. The proofs rely on an induction over the length of the traces of T and $[\![T]\!]_Z$ and imply an equivalence of the sets of traces of T and $[\![T]\!]_Z$. This implies $\mathcal{L}_T([\![T]\!]_Z) = \mathcal{L}_T(T)$. □

Theorem 16 (Correctness of Untimed NBA Construction)**.** *For an untimed NBA B_T derived from a TA T according to Definition 40 it holds that $\mathcal{L}(B_T) = \mathcal{L}_\mu(T)$.*

Proof. Assume a zone automaton $[\![T]\!]_Z = (S_Z, s^0_Z, \Sigma_Z, \Theta_Z)$ for a TA T and let $B_T = (S_B, \Sigma_B, \rightarrow, S^0_B, F)$ be the associated untimed NBA obtained according to Definition 40 with $[\![T]\!]_Z$.

Lemma 10 permits us to prove Theorem 16 by showing $\mathcal{L}(B_T) = \mathcal{L}_\mu([\![T]\!]_Z)$. We prove the conjecture by proving the stricter condition that the zone automaton $[\![T]\!]_Z$ and the untimed NBA B_T are untimed bisimilar. We may then conclude language equivalence by Lemma 9 and bisimilarity. It is appropriate to compare an untimed NBA B_T to a zone automaton $[\![T]\!]_Z$ by untimed bisimulation since B_T is an untimed automaton, in particular, an untimed zone automaton.

We now show that $R = \{((l, z_Z), (l, z_B)) | z^\uparrow_Z = z^\uparrow_B \wedge (l, z_Z) \in S_Z \wedge (l, z_B) \in S_B\}$ is an untimed bisimulation relation. R is an untimed bisimulation relation for the alphabet $\Sigma = (\Sigma_B \setminus \{\tau\}) \cup (\Sigma_Z \setminus \{\delta\})$, where the delay transitions δ and τ are removed by the construction of the untimed bisimulation.

We show for an arbitrary $((l, z_Z), (l, z_B)) \in R$ that $((l', z'_Z), (l', z'_Z)) \in R$ for all actions $a \in \Sigma$. $z_B \subseteq z_Z$ holds by construction of B_T since transitions in B_T reach a state after taking an action transition. Hence, there are two cases to be considered in order to determine whether $z^\uparrow_Z = z^\uparrow_B$:

1. If $z_Z = z_B$, then the untime construction creates a transition $(l, z_B) \overset{a}{\rightsquigarrow} (l', z'_Z) \in \rightarrow$ iff $(l, z_Z) \xrightarrow{\delta} (l, z^\uparrow_Z) \xrightarrow{a} (l', z'_Z) \in \Theta_Z$ exists.

2. If $z_Z = z^\uparrow_B$, then the untime construction creates a transition $(l, z_B) \overset{a}{\rightsquigarrow} (l', z'_Z) \in \rightarrow$ iff $(l, z_Z) \xrightarrow{a} (l', z'_Z) \in \Theta_Z$ exists.

In both cases it holds that $((l', z'_Z), (l', z'_Z)) \in R$. Thus, R is a bisimulation relation. The tuple $((l_0, z_0), (l_0, z_0))$ of the initial states of $[\![T]\!]_Z$ and B_T is in R, since $S^0_B = \{s^0_Z\}$ holds by the definition of the untime construction, and trivially $z_0^{\uparrow} = z_0^{\uparrow}$. This implies that B_T and $[\![T]\!]_Z$ are untimed bisimilar and $\mathcal{L}(B_T) = \mathcal{L}_\mu([\![T]\!]_Z)$ holds. □

Equivalence Check for Untimed Languages. Given that the zone automaton construction delivers closed NBAs we can reduce the admissibility test $\mathcal{L}_\mu(T) = \mathcal{L}_\mu(T_\iota)$ defined over infinite languages to an equivalence test over the finite prefixes of these languages, represented by interpreting the zone automata as NFAs. The following theorem justifies this reduction.

Theorem 17 (Language Equivalence of Closed NBAs)**.** *Given closed NBAs B and B', if $\mathcal{L}_{\mathrm{f}}(B) = \mathcal{L}_{\mathrm{f}}(B')$ then $\mathcal{L}(B) = \mathcal{L}(B')$.*

Proof. For a closed NBA B, it is easy to see that $\mathcal{L}_f(B) = \mathit{pref}(\mathcal{L}(B))$. Assume $\mathcal{L}_f(B) = \mathcal{L}_f(B')$ and $\mathcal{L}(B) \neq \mathcal{L}(B')$. This means that, w.l.o.g., $(\exists v)(v \in \mathcal{L}(B) \wedge v \notin \mathcal{L}(B'))$. This implies that there is no accepting run on some v in B'. Since B' is closed, the only way of not accepting v is that B' blocks when reading a finite prefix $\hat{v} = v_1, \ldots, v_n$ of v. Hence, $\hat{v} \in \mathcal{L}_f(B) \wedge \hat{v} \notin \mathcal{L}_f(B')$, which contradicts $\mathcal{L}_f(B) = \mathcal{L}_f(B')$. Therefore, $\mathcal{L}(B) = \mathcal{L}(B')$. □

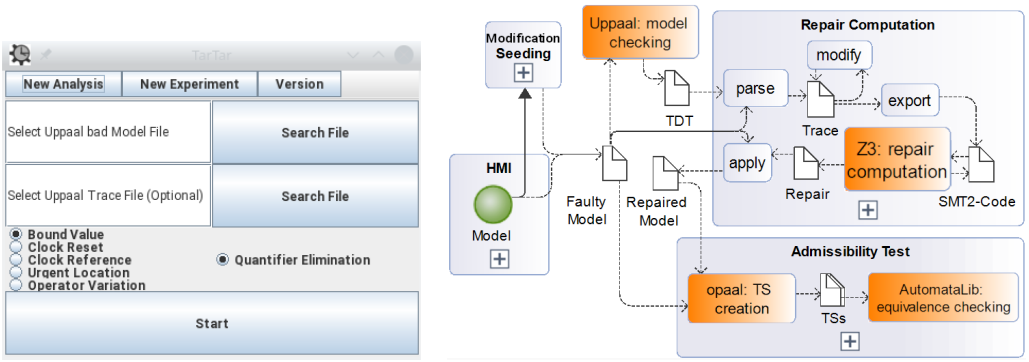
Complexity of the Admissibility Test. In order to test admissibility, we generate the state space of the original TA and the TA to which we apply the repair. The computation of the state space uses real-time model checking which is in PSPACE [6]. Afterwards, we interpret the two state spaces as NFAs and execute a language equivalence test. Checking language equivalence of two NFAs is decidable in NP [53]. The overall complexity of the admissibility test is thus in PSPACE.

Discussion. One may want to adapt the admissibility test so that it only considers divergent traces, e.g., in cases where only unbounded liveness properties need to be preserved by a repair. This can be accomplished as follows. First, an overapproximating non-zenoness test [14] can be applied to T and T_ι. If it shows non-zenoness, then one knows that the respective TA does not include convergent traces. If this test fails, a more expensive test needs to be developed. It requires a construction of the untimed NBA using the approach from [7], and subsequently a language equivalence test of the untimed languages accepted by the untimed NBAs using, for instance, the automata-theoretic constructions proposed in [35]. Checking language equivalence of two NBAs is in PSPACE [37]. Creating the untimed NBA, which is the input to the equivalence check, is also in PSPACE. In conclusion, the overall complexity of this analysis remains in PSPACE.

10.4 Tool Implementation

We implemented the repair computations and the admissibility test in the tool TarTar. The software architecture of TarTar is depicted in Figure 10.2(b). The

orange rectangles in the figure represent external tools that TarTar calls in the course of the repair analysis. UPPAAL is a closed-source model checking tool, which TarTar uses in order to compute a TDT for a given model and property. The SMT solver Z3 [42] is used to solve the generated partial MaxSMT problems. In order to check the admissibility of a repair, TarTar uses the tool opaal [39] and the AutomataLib component of LearnLib [75], since they conveniently provide functionality used in the implementation of the admissibility test.



(a) TarTar GUI (b) TarTar Architecture

Figure 10.2: TarTar Tool

Data Flow Architecture. TarTar consists of many computation steps. For example, a TDT is parsed internally and stored as a *Trace*. This *Trace* is then modified and exported as SMT-LIB2 [16] code. We define a computation step of TarTar as the computation transforming input into result artifacts. This focus on artifacts ensures a highly cohesive architecture and clear interfaces between any two computation steps. Computation steps with identical objectives are grouped into a project. This results in four projects depicted by blue rectangles in Figure 10.2(b).

- *HMI* denotes the user interfaces of TarTar. The user inputs a timed model. TarTar then calls the project *Repair Computation* using a faulty timed model as a parameter. In case that the model is correct, TarTar calls the project *Modification Seeding*.
- *Modification Seeding* seeds modifications into a correct model and then repairs the resulting faulty models by computing repairs using *Repair Computation*. We use this analysis in Section 10.5 in order to benchmark the *Repair Computation* analyses.
- *Repair Computation* computes candidate repairs for a faulty timed model, applies these repairs to the model and finally automatically calls the *Admissibility Test*.
- *Admissibility Test* checks for every repaired model whether the computed repair is admissible.

Control Flow Architecture. TarTar iteratively computes a set of repairs for a given faulty UPPAAL model and a given property Π using the following steps:

0. *Counterexample Creation.* TarTar calls UPPAAL to verify the model against Π. In case Π is violated, it stores a shortest symbolic TDT witnessing the violation in XML format.

1. *Diagnostic Trace Creation.* TarTar parses the model and the TDT into a data structure *Trace*. To add potential repairs, TarTar copies the trace and replaces the constraints that will potentially be subject to a repair by their modified variants. The modified trace is then translated to a logic constraint system $\mathcal{T}^*$, represented in SMT-LIB2 code.

2. *Repair Computation.* Z3 [42] then solves a MaxSMT problem on the modified trace constraint system, computing a repair in which the number of unmodified constraints on the variation variables of the form $v^* = 0$ is maximized. Since Z3 can solve a MaxSMT problem only for quantifier-free linear real arithmetic, TarTar first runs a quantifier elimination on the constraint system $\forall \delta_j, c_j . \mathcal{T}^* \Rightarrow \Phi$ of $\mathcal{F}^*_H$.

3. It then solves the MaxSMT problem with soft constraints requiring $v^* = 0$ for all variation variables. In case no solution is found, TarTar terminates. Otherwise, TarTar applies the repair to the faulty model and returns a repaired model.

4. *Admissibility Test.* TarTar checks the admissibility of a repair and compares the untimed languages of the faulty and repaired models. TarTar calls the model checker opaal in order to compute the timed transition systems (TTS) of the original and the repaired UPPAAL model. We modified the opaal model checker in such a way that it returns the TTS for a model. TarTar then checks whether the two TTS have equivalent untimed languages, in which case the repair is admissible. This check is implemented using the library AutomataLib. In case the two TTS are not equivalent, the admissibility test returns a trace as a witness for the difference.

5. *Iteration.* TarTar enumerates all repairs, i.e., every combination of the same kind of constraint modifications that corrects the TDT. The repairs are iteratively enumerated starting with the ones that require the smallest number of modifications to the model. After a repair is computed, the combination of modified variables that has been found is prevented from being reconsidered for future repairs by setting these modification variables to 0 using hard asserts. TarTar then proceeds with attempting to compute further, previously unconsidered repairs.

Component Architecture. We implemented TarTar with the general infrastructure depicted in Figure 10.3. The interface *Job* provides a general abstraction for an algorithm and specifies the necessary input and result values of the algorithm by the class *Description*. TarTar contains a *Job* for the projects *Modification*

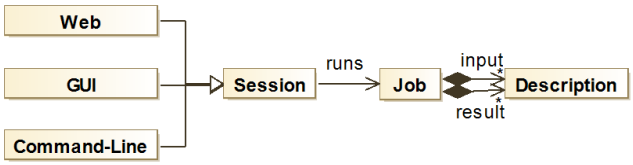


Figure 10.3: TarTar Component Architecture

Seeding, *Repair Computations* and *Admissibility Test*. The class *Session* executes a *Job* and derivations of *Session* provide the different interfaces to the user. With this infrastructure, the analysis implementation in TarTar is independent from the implementation of the user interfaces, thus reducing coupling and improving modifiability of the code.

Implementation Details. We implemented the different projects that constitute TarTar in Java and use the build-management tool maven [9] to manage the dependencies between the projects. TarTar interacts differently with the external tools that are needed for different purposes. It calls UPPAAL via the command-line interface in order to generate a TDT, calls Z3 via its API to compute a repair. For the admissibility test, it calls opaal using a command-line script and the AutomataLib as an included Java library. For the implementation of the TarTar analyses the following two details are essential.

- We modify constraints in an UPPAAL model in order to apply a repair or to seed a fault. Since neither clock constraints nor transitions possess explicit unique identifiers in an UPPAAL model, it is not obvious which constraint to change. We therefore uniquely identify a constraint by traversing the constraints in the sequence stored in the UPPAAL model file and use the constraint index in this sequence as its identifier.
- The complexity of the algorithms for solving quantifier elimination and the MaxSMT problem rises with the number of variables in the SMT model. We therefore reduce the number of variables by exploiting implied equality constraints. For example, a variable c_j is created for every clock c in every step j of the TDT. We eliminate c_j explicitly before quantifier elimination by replacing it with the term $\sum_{i \in r \ldots j} d_i$, where d_i is the time delay at step i in the trace and r is the last step before j where c was reset.

10.5 Case Studies and Experimental Evaluation

We evaluate the repair analyses by applying TarTar to the database model in Example 3 and to more complex case studies of different sizes.

10.5.1 Database Model Repairs

We applied TarTar to the database model in Figure 10.1. TarTar finds two admissible clock bound repairs. A replacement of $w \leq 2$ by $w \leq 1$ and a replace-

ment of $y \leq 1$ by $y \leq 0$ and $y \geq 1$ by $y \geq 0$ repairs the modified database model in Figure 10.1(c). With operator variation repair analysis, TarTar finds two admissible repairs that replace the operator in the clock constraint w >= 1 by < or <=, respectively. With clock reference repair analysis, TarTar finds 13 admissible clock reference modification repairs, each involving two modifications. Nine repairs replace y in the constraints $y \leq 1$ and $y \geq 1$ by a selection from the set of clocks z, x and w. Four repairs replace y in the constraint $y \leq 1$ by w or x, and w in the constraint $w \geq 1$ by y or z. Applying the reset repair analysis, TarTar finds four admissible repairs. One repair removes the reset of clock y, another removes the reset of clock z, and two repairs add a reset of clock x either on the transitions towards the location *reqProcessing* or the transition towards the location *serReceiving*. Applying the urgency location repair analysis, TarTar finds only two inadmissible and no admissible repairs, one setting the location *reqAwaiting* and the other the location *serReceiving* to urgent.

The computed repairs are reasonable and we expected them following a thorough manual inspection of the model. It remains to be evaluated whether TarTar can analyze more complex models.

10.5.2 Evaluation Strategy

In order to perform an experimental evaluation of the repair analyses, both qualitatively and quantitatively, we need a set of timed automata models that violate given properties. Such models are not publicly available in significant numbers and to the best of our knowledge, no benchmark suite for property violating timed automata models exists. As a consequence, the evaluation of our analyses is based on ideas taken from mutation testing [79]. Mutation testing evaluates a test set by systematically modifying the program code to be tested and computing the ratio of modifications that are detected by the test set. In adopting this strategy, we seed modifications in existing models and check whether those can be successfully repaired by our automated repair analyses. Thereby, we evaluate the quality of our repair technique.

Modification Seeding modifies a given TA model syntactically in one of the following ways:

- replacing the comparison operator in a clock constraint by $\{<, \leq, =, \geq, >\}$,
- swapping a clock name referenced in a clock constraint or invariant with some other clock name occurring in the original model,
- modifying the set of clocks that are reset in any given transition,
- switching for any location whether it is urgent or not,
- or modifying the bound of a single clock constraint by an amount selected from the set $\{-10, -1, +1, +0.1M, +M\}$, where M is the maximal clock bound occurring in a given model. Our observation was that making either a small modification that is close to the bound value or a modification in the order

of the maximal bound value M often introduces a property violation in the considered models.

The result is a modified TA that contains a single modification. TarTar automatically checks for every modified TA whether it violates a given property, in which case a TDT is generated.

Experimental Procedure. TarTar applies modification seeding to several models used for experimentation and generates a set of TDTs by performing model checking for a given property on each of the modified models. For every computed TDT, TarTar performs the above defined repair analyses one after another, and measures the ratio of TDTs for which the analyses compute at least one repair.

10.5.3 Experiments

We have applied this evaluation strategy to eight UPPAAL models (see Table 10.1). Not all of the models that we considered have been published with a property that can be violated by mutating a clock constraint. For those models, we manually identify a suitable timed safety property specifying an invariant condition which can then be violated by some of the proposed mutations. In particular, we add a property to the Bando [133] model which ensures that, for as long as the sender is active, its clock never exceeds the value of 28,116 time units. In the FDDI token ring protocol [133], the property that we use checks whether the first member of the ring never remains in any given state for more than 140 time units. The Viking model is taken from the set of test models of opaal [119]. For this model we use a property that checks whether one of the Viking processes can only enter a safe state during the first 60 time units. Note that all of these properties are satisfied by the original models prior to modification seeding.

We applied modification seeding to the models given in Table 10.1 and analyzed the obtained TDTs using the above described repair analyses implemented in TarTar. All analyses were performed on a computer with an i7-6700K CPU

Model	db rep.	csma	elevator	viking	bando	Pacemaker	SBR	FDDI
#Seed	110	191	88	310	1,955	1,187	353	314
#TDT	13	10	5	9	40	12	50	36
Time_{UP}	0.016	0.012	0.011	0.015	0.111	0.022	0.027	0.014
Length	4	2	1	18	279	9	84	11
#Repair	229	70	7	9	4,061	62	751	166
#Admissible	138(60%)	26(37%)	5(70%)	7(78%)	209(5%)	19(31%)	660(88%)	105(63%)
#Solved	9(70%)	8(80%)	4(80%)	5(56%)	21(53%)	10(83%)	31(62%)	34(94%)
Time_{QE}	89.346	0.049	0.049	86.539	31.555	0.663	117.057	29.859
#Timeout	2	0	0	21	46	20	86	51
$\text{Time}_{\text{Repair}}$	0.911	0.023	0.020	1.436	4.922	0.325	2.686	3.074
$\text{Mem}_{\text{Repair}}$	14.53	0.58	0.53	20.07	20.86	2.59	37.16	9.70
#Variable	30	16	6	120	1,156	116	765	116
#Constraint	91	72	28	180	8,144	988	1,211	272
Time_{Adm}	2.080	1.825	1.665	1.952	19.57	1.994	138.004	2.241
Mem_{Adm}	45	75	17	543	1,251	206	211	128

Table 10.1: Experimental Results According to Model.

Repair	Bound Mod.	Operator Var.	Clock Ref.	Reset Clock	Urgent Loc.
#Repair	533	3,929	693	45	155
#Admissible	364 (68%)	96 (2.4%)	625 (90%)	37 (82%)	47 (30%)
#Solved	85 (48%)	51 (29%)	35 (20%)	13 (7%)	37 (21%)
Time_{QE}	15.209	117.057	33.282	89.346	0.107
#Timeout	8	44	61	113	0
$\text{Time}_{\text{Repair}}$	4.922	2.686	3.074	0.911	0.135
$\text{Mem}_{\text{Repair}}$	20.86	37.16	14.13	14.53	3.16
#Variable	1,156	996	1,120	996	1,120
#Constraint	2,498	8,144	5,355	2,836	2,502
Time_{Adm}	138.004	59.117	116.944	2.051	58.551
Mem_{Adm}	525	543	206	45	1,251

Table 10.2: Experimental Results According to Type of Repair.

(4.00GHz), 60GB of RAM and a 64 bit Linux operating system. We summarize the results of this experiment per considered model (Table 10.1) and per type of considered repair (Table 10.2). In Table 10.3, we give insight into a subset of the repair analysis results for the cases where the repair analysis was of the same type as the seeded modification.

Row *Seed* in the tables contains the count of seeded modifications that result in a number $\#TDT$ of property violating models. $Time_{UP}$ is the maximal time that UPPAAL needs to create a TDT for the property violating models, and the longest TDT has a length of *Length*. TarTar computed a number *#Repair* of repairs for the TDTS, of which *#Admissible* are admissible. We say a TDT is solved when at least one admissible repair is computed by a repair analysis. *#Solved* states the number of solved TDTs. The computation effort for a repair analysis is given by the time $Time_{QE}$ for successful quantifier elimination, the number of timeouts $\#Timeout$ that occurred during quantifier elimination after 10 minutes, the average time $Time_{Repair}$ to compute a repair, and the memory consumption Mem_{Repair} for the overall analysis. The constraint system that Z3 solves has the count $\#Variable$ of variables and $\#Constraint$ of constraints. The effort for the admissibility test is given in time $Time_{Adm}$ and memory Mem_{Adm}. All times are given in seconds and memory consumption in MB. Notice that we omit the lines pertaining to the modification seeding and TDT computation in Table 10.2 since they are irrelevant there.

We illustrate the evaluation procedure using an instance of the pacemaker case study, which is a real world model of realistic size. One of the seeded bound modifications increases a location invariant of this model which controls the minimal heart period to remain within a range from 400 to 1,600 time units. The modification allows the pacemaker to delay an induced ventricular beat for too long so that this violates the property that the time between two ventricular beats of a heart is never longer than the maximal heart period of 1,000. TarTar finds three repairs. The first repair reduces the maximal time delay between two articular heartbeats such that the pacemaker cannot and no longer needs to trigger the articular heartbeat. The second repair reduces the maximal time delay between two ventricular heartbeats such that the pacemaker cannot trigger the first ventricular heartbeat. Both repairs change the heart behavior and remove functional behavior of the pacemaker and, hence, are classified as inadmissible. In the model context, this appears to be

Seed/Repair	Bound Mod.	Operator Var.	Clock Ref.	Reset Clock	Urgent Loc.
#Seed	1,385	1,385	578	891	269
#TDT	60	72	27	8	8
Time_{UP}	0.111	0.083	0.093	0.027	0.021
Length	279	248	279	84	18
#Repair	314	3,508	383	6	11
#Admissible	226 (72%)	68 (1.9%)	380 (98%)	6 (100%)	2 (18%)
#Solved	53 (88%)	38 (52%)	9 (33%)	1 (12%)	1 (12%)
Time_{QE}	15.209	31.858	25.392	43.922	0.027
#Timeout	3	8	10	7	0
$\text{Time}_{\text{Repair}}$	4.922	1.436	2.525	0.558	0.049
$\text{Mem}_{\text{Repair}}$	20.86	12.12	13.01	10.88	0.71
#Variable	1,156	996	1,120	25	90
#Constraint	2,498	8,144	5,355	57	279
Time_{Adm}	18.045	52.749	116.944	2.051	19.570
Mem_{Adm}	525	543	205	33	1,251

Table 10.3: Subset of Experimental Results where Type of Repair Corresponds to Type of Seeded Modification.

reasonable since the repairs restrict the environment of the pacemaker, and not the pacemaker itself. The third repair is admissible and reduces the bound modified during the seeding of bound modifications by 1199.5. The minimal heart period is then also below or equal to the maximal heart period of 1,000.

TarTar seeded 4,508 modifications. This resulted in 175 TDTs in total. 60 TDTs were due to bound modification, 72 were due to operator variation, 27 were due to changing the clock reference, 8 were due to complementing the reset of clocks and 8 were due to the switching of urgent locations (see Table 10.3). In total, TarTar found $5,355$ repairs, out of which 1,169 were admissible. It found at least one admissible repair for 122 (69%) of the 175 TDTs. The maximal number of modified constraints in an admissible repair computed for a single TDT using all types of repair analysis was 25. A total of $4,222$ repairs were of the same type as the seeded modification. Out of these same type repairs, 682 were admissible. At least one admissible same type repair exists for 102 (58%) of the 175 TDTs.

10.5.4 Result Interpretation

Pacemaker Instance Results. Our repair strategy minimizes the number of repairs but does not optimize the computed value. For instance, in the pacemaker model the computed repair of 1199.5 time units would be a correct and admissible repair even if the value was reduced by 600 time units, which would be the minimal possible repair value.

Modification Seeding Results. Few of the seeded modifications resulted in a property violation. TarTar seeded 4,508 faults, which led to 175 TDTs, thus only 3.9% of these modifications result in a TDT. This supports that, in practice, often only a few time constraints have an impact on the satisfaction of a verified property.

Repair results in which a seeded modification is of arbitrary type. TarTar computes at least one admissible repair by bound modification for 85 (48%), by operator

variation for 51 (29%), by clock reference for 35 (20%), by clock reset for 13 (7%) and by urgent location for 37 (21%) of the 175 TDTs. Every analysis on its own computes less admissible repairs than the combination of all repair analyses, which solves 122 (69%) of the 175 TDTs. The largest number of constraints that an admissible repair modified was 25, which is less than anticipated. This low number of modified constraints allow us to infer that only few constraints have an impact on whether a property is violated or not.

TarTar does not find an admissible repair for every TDT. For instance, TarTar finds an admissible repair for 53 TDT of the 60 TDTs created by bound modification, but only 3 analyses result in a timeout (Table 10.3). Hence, there exist 4 TDTs for which no admissible repair was computed even though they did not result in a timeout. There are two reasons for this phenomenon. Notice that a TDT is caused by modification seeding in which a constraint c in a model is modified. First, the seeding of a modification in the model may enable or disable transitions and, as a consequence, change the functional behavior of the unmodified model. A repair computed by TarTar may not precisely undo the modification, and therefore it cannot be guaranteed that the repaired model is functionally consistent with the unmodified model. Second, even if the modified model is functionally consistent with the unmodified model, the computed repair may not undo the modification and at the same time be inadmissible.

Computational effort. A comparison of Time_{QE} and $\text{Time}_{\text{Repair}}$ in every table confirms that the computational effort for the repair computation is largely determined by the quantifier elimination step, as we also concluded from the complexity analysis for the repair analysis. Only the urgent location repair consumes more computation time during the repair computation than during the quantifier elimination step. We expect that in light of the observed 226 timeouts, a more efficient quantifier elimination procedure would lead to a significantly higher number of repairs that could be computed without encountering a timeout. Furthermore, the number of timeouts, and thus the computation time needed for the repair, seems to correlate with the length of the analyzed TDT. With 86 the *SBR* model includes the largest number of timeouts and the third longest TDT with a length of 84 steps. The *bando* model has the third most timeouts (46) and the longest TDT. Obviously, the longer the TDT, the larger the resulting constraint system, leading to increased computational effort. With $1,156$ variables and $8,144$ constraints, the *bando* model yields the largest constraint system. The *SBR* model has the second largest constraint system with 765 variables and $1,211$ constraints. The model *FDDI* has a shorter TDT with a length of 11 transitions and a much smaller constraint system with 116 variables and 272 constraints. In order to provide some statistical evidence for the impact of the TDT length and the intrinsic model complexity on the computational effort we analyzed the statistical relationship between the length of the TDT and the computation time for a repair ($T_r = Time_{QE} + Time_{Repair}$) as well as the relationship between $\#Variable$ and T_r, both by estimating Kendall's tau [50] for every TDT. Kendall's tau is a measure for the ordinal association between two measured quantities. A correlation computed by Kendall's tau estimation approach is considered significant if the probability p that there is actually no correlation in a larger

data set is below a certain threshold. We use the commonly applied significance threshold of 0.05. The length of a TDT is significantly related ($\tau_1 = 0.521$, $p < .001$) to T_r. Also *#Variable* is significantly related ($\tau_2 = 0.496$, $p < .001$) to T_r. From this we conclude that the complexity of a repair depends not only on the TDT length, but also on the intrinsic complexity of the model as expressed by the number of variables occurring in the model.

We also observe that the admissibility test requires more computational resources than the repair computation. The maximum amount of memory used for the admissibility test was 1,251MB in contrast to 37.16MB for the repair computation. This is in line with our expectation since the admissibility test involves searches of the state space of the full NTA, while the repair analyses only consider a single TDT.

Modifying states from urgent to non-urgent during modification seeding resulted in only 8 TDTs. This low number is due to the observation that the models considered contain only very few urgent states. Modifying non-urgent states to urgent ones, however, did not lead to a single property violation resulting in a TDT. The rationale is that urgency ensures to leave a state immediately without a delay which leads to a restriction rather than a relaxation regarding the time budget spent along an execution trace. As a consequence, making a state urgent does not cause a property violation in many models since the type of the checked properties is typically time bounded reachability, and a restricted time budget does not make it more likely that the property is violated.

Results of repairs that have the same type as the seeded modification. Recall that we seed different types of modifications that correspond to the different types of repair analyses that we propose. For every type of a repair analysis, we now compare the probability computing an admissible repair when the seeded modificaton is of the same type as the repair (see Table 10.3) with the case when the seeded modification is of arbitrary type (see Table 10.2). Bound modification analysis repairs a TDT with a seeded bound modification with a higher probability (88% to 48%) than a modification of an arbitrary type. Also, operator modification analysis (52% to 29%), clock reference analysis (33% to 20%) and clock reset analysis (12% to 7%) compute a repair with a higher probability for the same type of seeded fault as a modification of an arbitrary type. Only urgent location analysis computes a repair with a lower probability (12% to 21%) for a TDT with a seeded urgent location modification than for a modification of arbitrary type. In summary, with the exception of the urgent repair analysis, an admissible repair is computed with a higher probability if the repair analysis is of the same type as the seeded modification.

We finally observe that the repair computations of a seeded modification of an arbitrary type (Table 10.2) and repair computations of the same type as the seeded modification (Table 10.3) require similar computational effort in terms of time and memory consumption. We conclude that the computational effort of a repair analysis is independent of the type of a seeded modification that it repairs.

10.5.5 Limitation of the Repair Analyses

We applied the repair analyses also to the Fischer's models from Chapter 9.5. The Fischer's model reaches a property violation because of a wrong time constraint. The resulting TDT depicted as a sequence diagram in Figure 9.2(b). The TDT describes a transition sequences where every realization violates the given property. Since no good realization of the TDT exists, the repair analyses do not compute any repair for this TDT. Remember, that a repair also represents a static cause. From a cause computation perspective it is reasonable that no static cause exists in the Fischer's model since the good realization is necessary to satisfy the counterfactual argument. A future extension might can either use suffix-blocking partial realization or compares the TDT with a property satisfying timed traces of the model. Both extension compare the TDT with good realizations and would enable the analyses to compute causes and, hence, further repairs.

10.6 Conclusion

We presented an approach to derive minimal repairs for timed reachability properties of NTA models from TDTs returned as counterexamples during model checking. The objective of the presented analysis was to compute static causes. It turned out that the static cause computations are also a repair syntheses that facilitates fault localization and debugging of such models during the design process. Our approach includes several repair analyses based on MaxSMT solving, an admissibility criterion for the computed repairs, and the development of a repair tool. We evaluated the repair analyses by a number of case studies of realistic complexity, and presented a limitation. We have observed that our analyses computes a significant number of admissible repairs within realistic computation time bounds and memory consumption.

Part IV

Conclusion

We present methods for explaining and repairing failures in untimed and timed systems based on causality.

Causality Checking is an existing approach to enumerate causes for violations of regular safety properties in untimed systems. We enhanced Causality Checking in multiple ways.

1. We present complete cause definitions.

2. We propose the CTBS algorithm to compute every causal trace in a transition system. The CTBS algorithm, when run together with a BFS state space search, ensures that it computes correct causes up to the current search depth. The advantage is that the causes are correct even when the overall analysis timeouts for a very large model.

3. We further compute the action orders of causes by analyzing the strict partial order in a causal trace set and depict this order in a fault tree. The representation of action orders in causes provides helpful information to understand the occurrence of a failure.

We also showed by a case study on autonomous driving that Causality Checking can support the safety analysis of a safety-critical system. The analysis automates the safety analysis of different software to hardware mappings. Additionally, we proposed another analysis to assess the fail-operational behavior which is of special interest in the context of autonomous driving.

Real-time is an important aspect for the correctness of many embedded systems. The real-time behavior of a system is not only described by concrete transitions but also by time delays in locations of the system. A real-time model checker can find violations in a real-time system and returns a TDT that leads to this violation. We proposed two alternative approaches to analyze the time aspect in a TDT.

- A causal range analysis compares the possible time delays in a TDT and returns causal ranges where every execution satisfying the causal range reaches a property violation. A causal range explains which time delays are responsible for the TDT to occur.

- The repair analysis computes syntactic modifications of syntactic time constraints that prevent the TDT and represent repairs for the underlying real-time model. We also proposed an admissibility test that checks whether a found repair does not change the functional behavior of the real-time system. In the evaluation, the repair analysis computed an admissible repair for 69-88% of the TDTs.

A causal range gives insight into which time delays impact the satisfaction of a property, while a repair hints at wrong time constraints. Both are causes by the counterfactual argument, give insight into why a failure occurred, and hence can support an engineer to chose the right timing in the design space of a real-time system.

The repair analysis falsely gives the appearance that a cause is a repair. Causes and repairs are different objects since in Chapter 9.5, we compute dynamic causes

for the Fischer's model that are not repairs. While a cause reasons about the occurrence of an effect, a repair modifies a model in a way that some part of the cause cannot occur and, hence, the effect does not occur. We see that a static cause and a repair are closely related because if we cannot compute a cause, we also cannot compute a repair. A static cause is useful since it is both a dynamic cause and a syntactic modification of a model.

A combination of Causality Checking and time analyses can analyze the functional behavior and the time behavior of a real-time system. For this purpose, untime the TDTs of a real-time model and compute causes by applying Causality Checking on the resulting action traces. By this construction, for each cause exists TDTs that untimed satisfy the cause. The time behavior of these TDTs can now be analyzed by one of the time analyses presented above.

In this thesis, we presented approaches that combine model checking and actual causality to explain failures in untimed and timed systems. Causality Checking enumerates the causes of a system because of which a property violation occurs, and can support the safety analysis of a system as demonstrated. Time analyses help to explore the infinite design space of real-time behaviors. As such, the approaches can automate the failure analysis of a system and are valuable to be integrated into an MDE process.

Future Work

A way to make Causality Checking more efficient is to use symbolic state space exploration techniques. In the evaluation, Causality Checking had timeouts for complex models and the results also show that the causes of big models have a large number of causal traces. Symbolic model checking may improve Causality Checking since it can efficiently store action traces with the same multi-set of actions that differ only in the action order.

The presented algorithms compute potential causes but does not compute the actual causes for a specific world. Following the approach of Halpern and Pearl [59], an interesting direction is to analyze for a specific world that satisfies several potential actual causes, the actual cause that led to the property violation. An analysis of contingencies is therefore necessary.

In future work we plan to explore the interplay between different repairs that are computed for a timed system that still violates a property, and develop refined strategies to select promising repairs from a repair set.

Another direction to support design space exploration is to not only compute clock constraint modifications for a faulty model, but also to compute relaxations of clock constraints in a correct model. A first step in this direction is to encode not only a single TDT but the complete model.

Further, the repair analysis can be generalized. Time in a real-time system changes linearly. A hybrid system can describe also other types of continuous changes. The findings in the repair analysis are promising to be applicable to the analysis of a hybrid system. Even more interesting is a combination of Causality Checking and one of the timed analyses to compute causes for hybrid systems.

List of Figures

List of Tables

Listings

Bibliography

[1] R. Adler, P. Feth, and D. Schneider. Safety Engineering for Autonomous Vehicles. In *DSN Workshops*, pages 200–205. IEEE Computer Society, 2016. 12

[2] H. Aljazzar and S. Leue. Directed Explicit State-Space Search in the Generation of Counterexamples for Stochastic Model Checking. *IEEE Trans. Software Eng.*, 36(1):37–60, 2010. 11

[3] H. Aljazzar and S. Leue. K*: A Heuristic Search Algorithm for Finding the k Shortest Paths. *Artif. Intell.*, 175(18):2129–2154, 2011. 11

[4] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987. 28

[5] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015. 13

[6] R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking in Dense Real-time. *Inf. Comput.*, 104(1):2–34, 1993. 139

[7] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. IV, VI, 136, 137, 139

[8] T. Ando, H. Yatsu, W. Kong, K. Hisazumi, and A. Fukuda. Formalization and Model Checking of SysML State Machine Diagrams by CSP#. In *ICCSA (3)*, volume 7973 of *Lecture Notes in Computer Science*, pages 114–127. Springer, 2013. 9

[9] Apache Software Foundation. *Maven*, 2019. https://maven.apache.org/. 142

[10] A. Avizienis, J. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. 3

[11] G. M. Bahig and A. El-Kadi. Formal Verification of Automotive Design in Compliance With ISO 26262 Design Verification Guidelines. *IEEE Access*, 5:4505–4516, 2017. 12

[12] C. Baier, C. Dubslaff, F. Funke, S. Jantsch, R. Majumdar, J. Piribauer, and R. Ziemek. From Verification to Causality-Based Explications (Invited Talk). In N. Bansal, E. Merelli, and J. Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16,*

2021, Glasgow, Scotland (Virtual Conference), volume 198 of *LIPIcs*, pages 1:1–1:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 10

[13] C. Baier, B. Haverkort, H. Hermanns, and J. P. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003. 26, 93

[14] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008. 3, 16, 17, 18, 19, 89, 134, 136, 139

[15] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *POPL*, pages 97–105. ACM, 2003. 9

[16] C. Barrett, P. Fontaine, and C. Tinelli. *SMT-lib*, 2017. http://smtlib.cs.uiowa.edu/language.shtml. 140

[17] V. Batusov and M. Soutchanski. Situation Calculus Semantics for Actual Causality. In *AAAI*, pages 1744–1752. AAAI Press, 2018. 10

[18] A. Bäuerle. Analysis and Repair for Real-Time Properties of SysML Models. Master's thesis, University of Konstanz, 2020. 4

[19] S. Beckers and J. Vennekens. A Principled Approach to Defining Actual Causation. *Synth.*, 195(2):835–862, 2018. 10

[20] A. Beer, S. Heidinger, U. Kühne, F. Leitner-Fischer, and S. Leue. Symbolic Causality Checking Using Bounded Model Checking. In *SPIN*, volume 9232 of *Lecture Notes in Computer Science*, pages 203–221. Springer, 2015. 10, 11

[21] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler. Explaining Counterexamples Using Causality. *Formal Methods in System Design*, 40(1):20–40, 2012. 10

[22] M. T. Befrouei, C. Wang, and G. Weissenbacher. Abstraction and Mining of Traces to Explain Concurrency Bugs. *Formal Methods Syst. Des.*, 49(1-2):1–32, 2016. 9

[23] S. Behere and M. Törngren. A Functional Reference Architecture for Autonomous Driving. *Information & Software Technology*, 73:136–150, 2016. 87, 89, 94

[24] H. Ben-Abdallah and S. Leue. Timing Constraints in Message Sequence Chart Specifications. In *FORTE*, volume 107 of *IFIP Conference Proceedings*, pages 91–106. Chapman & Hall, 1997. 106

[25] J. Bendík, A. Sencan, E. A. Gol, and I. Cerná. Timed Automata Relaxation for Reachability. In *TACAS (1)*, volume 12651 of *Lecture Notes in Computer Science*, pages 291–310. Springer, 2021. 13

[26] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995. 5, 29, 105, 106

[27] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003. 12, 26, 27, 107, 136, 137

[28] D. Beyer and T. Lemberger. Software Verification: Testing vs. Model Checking - A Comparative Evaluation of the State of the Art. In *Haifa Verification Conference*, volume 10629 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2017. III, V, 4

[29] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, J. Ouaknine, and J. Worrell. Model Checking Real-Time Systems. In *Handbook of Model Checking*, pages 1001–1046. Springer, 2018. 27, 28

[30] G. Caltais, S. Leue, and H. Singh. Correctness of an ATL Model Transformation from SysML State Machine Diagrams to Promela. In *MODELSWARD*, pages 360–372. SCITEPRESS, 2020. 4

[31] G. Caltais, M. R. Mousavi, and H. Singh. Causal Reasoning for Safety in Hennessy Milner Logic. *Fundam. Informaticae*, 173(2-3):217–251, 2020. 10

[32] O. Carrillo, S. Chouali, and H. Mountassir. Formalizing and Verifying Compatibility and Consistency of SysML Blocks. *ACM SIGSOFT Softw. Eng. Notes*, 37(4):1–8, 2012. 9

[33] A. Chapman. Camels, Diamonds and Conterfactuals: a Model for Teaching Causal Reasoning. *Teaching History*, pages 46–53, 09 2003. 124, 126

[34] G. Chartrand and L. Lesniak-Foster. *Graphs & Digraphs*. Chapman and Hall/CRC, Boca Raton [u.a.], 4. edition, 2005. 20

[35] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A Unified Approach for Showing Language Inclusion and Equivalence Between Various Types of omega-Automata. *Inf. Process. Lett.*, 46(6):301–308, 1993. 139

[36] P. Cuenot, C. Ainhauser, N. Adler, S. Otten, and F. Meurville. Applying Model Based Techniques for Early Safety Evaluation of an Automotive Architecture in Compliance with the ISO 26262 Standard. In *Proceedings of the 7th European Congress on Embedded Real Time Software and Systems (ERTS ^{2})*, 2014. 12

[37] D. B. Czerbo. Handbook of Theoretical Computer Science : J. van leeuwen, ed., vol. A: Algorithms and Complexity, Vol. B: Formal Methods and Semantics (Elsevier, Amsterdam, 1990), 2296 pp., hardcover, dfl. 555.00. *Artif. Intell. Medicine*, 4(4):309, 1992. 139

[38] A. R. da Silva. Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Comput. Lang. Syst. Struct.*, 43:139–155, 2015. 3

[39] A. E. Dalsgaard, R. R. Hansen, K. Y. Jørgensen, K. G. Larsen, M. C. Olesen, P. Olsen, and J. Srba. opaal: A Lattice Model Checker. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 487–493. Springer, 2011. 5, 105, 140

[40] J. H. Davenport and J. Heintz. Real Quantifier Elimination is Doubly Exponential. *J. Symb. Comput.*, 5(1/2):29–35, 1988. 122, 134

[41] M. de Jonge and T. C. Ruys. The SpinJa Model Checker. In *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 124–128. Springer, 2010. 64

[42] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. 120, 128, 140, 141

[43] C. Deng and P. Cousot. The Systematic Design of Responsibility Analysis by Abstract Interpretation. *ACM Trans. Program. Lang. Syst.*, 44(1):3:1–3:90, 2022. 11

[44] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, 1995. 11

[45] H. Dierks, S. Kupferschmid, and K. G. Larsen. Automatic Abstraction Refinement for Timed Automata. In *FORMATS*, volume 4763 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007. 12

[46] R. Dimitrova, R. Majumdar, and V. S. Prabhu. Causality Analysis for Concurrent Reactive Systems (Extended Abstract). In *CREST@ETAPS*, volume 286 of *EPTCS*, pages 31–33, 2018. 12

[47] S. Edelkamp and S. Schrödl. *Heuristic Search - Theory and Applications*. Academic Press, 2012. 11

[48] D. Eppstein and J. A. Simons. Confluent Hasse Diagrams. *J. Graph Algorithms Appl.*, 17(7):689–710, 2013. 11

[49] M. Ergurtuna, B. Yalcinkaya, and E. A. Gol. An Automated System Repair Framework with Signal Temporal Logic. *Acta Informatica*, pages 1–27, 2021. 13

[50] A. Field. *Discovering Statistics Using IBM SPSS Statistics: and Sex and Drugs and Rock 'n' Roll, 4th Edition*. Sage, 2013. 147

[51] E. Fredkin. Trie Memory. *Commun. ACM*, 3(9):490–499, Sept. 1960. 17

[52] J. Fritzsch, T. Schmid, and S. Wagner. Experiences from Large-Scale Model Checking: Verifying a Vehicle Control System with NuSMV. In *ICST*, pages 372–382. IEEE, 2021. 12

[53] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 139

[54] M. Ghadhab, S. Junges, J. Katoen, M. Kuntz, and M. Volk. Model-Based Safety Analysis for Vehicle Guidance Systems. In *SAFECOMP*, volume 10488 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2017. 12, 97

[55] G. Goessler and L. Astefanoaei. Blaming in Component-Based Real-Time Systems. In *EMSOFT*, pages 7:1–7:10. ACM, 2014. 12

[56] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, 2006. 9

[57] A. Groce and W. Visser. What Went Wrong: Explaining Counterexamples. In *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2003. 9

[58] N. Hall. Two Concepts of Causation. In J. Collins, N. Hall, and L. Paul, editors, *Causation and Counterfactuals*, pages 225–276. MIT Press, 2004. 10

[59] J. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *The British Journal for the Phil. of Science*, 2005. 4, 10, 20, 21, 154

[60] J. Y. Halpern. A Modification of the Halpern-Pearl Definition of Causality. In *IJCAI*, pages 3022–3033. AAAI Press, 2015. 4

[61] J. Y. Halpern. *Actual Causality*. MIT Press, 2016. 9, 10, 12, 21

[62] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach - part II: explanations. In *IJCAI*, pages 27–34. Morgan Kaufmann, 2001. 10, 15

[63] J. Y. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach - Part II: Explanations. *The British Journal for the Philosophy of Science*, 2005. 113

[64] T. Han, J. Katoen, and B. Damman. Counterexample Generation in Probabilistic Model Checking. *IEEE Trans. Software Eng.*, 35(2):241–257, 2009. 11

[65] H. Hansen and A. Kervinen. Minimal Counterexamples in O(n log n) Memory and O(nˆ2) Time. In *ACSD*, pages 133–142. IEEE Computer Society, 2006. 11

[66] L. S. Heath and A. K. Nema. The Poset Cover Problem. *Open Journal of Discrete Mathematics*, 3(3), 2013. 11

[67] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980. 11

[68] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Inf. Comput.*, 111(2):193–244, 1994. 12, 136

[69] G. J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004. 49

[70] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation, Second Edition*. Addison-Wesley, 2000. 136

[71] X. Huang, Q. Sun, J. Li, and T. Zhang. MDE-Based Verification of SysML State Machine Diagram by UPPAAL. In *ISCTCS*, volume 320 of *Communications in Computer and Information Science*, pages 490–497. Springer, 2012. 4

[72] IBM Corp. IBM SPSS Statistics for Windows, Version 27, 2020. `https://www.ibm.com/analytics/spss-statistics-software`. 82

[73] A. Ibrahim, T. Klesel, E. Zibaei, S. Kacianka, and A. Pretschner. Actual Causality Canvas: A General Framework for Explanation-Based Socio-Technical Constructs. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2978–2985. IOS Press, 2020. 10

[74] A. Ibrahim and A. Pretschner. From Checking to Inference: Actual Causality Computations as Optimization Problems. In *ATVA*, volume 12302 of *Lecture Notes in Computer Science*, pages 343–359. Springer, 2020. 10

[75] M. Isberner, F. Howar, and B. Steffen. The Open-Source LearnLib - A Framework for Active Automata Learning. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015. 140

[76] ISO. Road vehicles - Functional Safety. ISO 26262, International Organization for Standardization, Geneva, Switzerland, 2011. III, V, 4, 5, 85, 88, 89, 90

[77] ISO. Draft International Standard, Road vehicles – Funtional Safety. Technical Report ISO/DIS 26262, International Organization for Standardization, Geneva, Switzerland, 2016. 86, 88, 89

[78] ISO. Road vehicles – Safety of the Intended Functionality. ISO ISO/PAS 21448, International Organization for Standardization, Geneva, Switzerland, 2019. 88

[79] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011. 143

[80] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2012. 3, 125, 126, 129

[81] M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 504–509. Springer, 2011. 13

[82] V. Kabilan. Contract Workflow Model Patterns Using BPMN. In *EMMSAD*, volume 363 of *CEUR Workshop Proceedings*, pages 171–182. CEUR-WS.org, 2005. 49

[83] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Comb.*, 4(4):373–396, 1984. 122, 134

[84] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikucionis, and P. Olsen. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In *FMICS*, volume 9128 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2015. 12

[85] A. Knapp, S. Merz, and C. Rauh. Model Checking - Timed UML State Machines and Collaborations. In *FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002. 9

[86] M. Kölbl and S. Leue. Automated Functional Safety Analysis of Automated Driving Systems. In *FMICS*, volume 11119 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2018. VI, 7, 85

[87] M. Kölbl and S. Leue. An Efficient Algorithm for Computing Causal Trace Sets in Causality Checking. In *ATVA*, volume 11781 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2019. 7, 49

[88] M. Kölbl and S. Leue. An Algorithm to Compute a Strict Partial Ordering of Actions in Action Traces. In *ISoLA (4)*, volume 12479 of *Lecture Notes in Computer Science*, pages 10–26. Springer, 2020. 7, 71, 127

[89] M. Kölbl, S. Leue, and R. Schmid. Dynamic Causes for the Violation of Timed Reachability Properties. In *FORMATS*, volume 12288 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2020. 7, 105, 113

[90] M. Kölbl, S. Leue, and H. Singh. From SysML to Model Checkers via Model Transformation. In *SPIN*, volume 10869 of *Lecture Notes in Computer Science*, pages 255–274. Springer, 2018. 4

[91] M. Kölbl, S. Leue, and T. Wies. Clock Bound Repair for Timed Systems. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2019. 7, 105, 123, 126, 127

[92] M. Kölbl, S. Leue, and T. Wies. TarTar: A Timed Automata Repair Tool. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 529–540. Springer, 2020. 7, 105, 127

[93] M. Kölbl, S. Leue, and T. Wies. Automated Repair for Timed Systems. *submitted for journal publication*, 2021. 7

[94] P. Koopman and M. Wagner. Challenges in Autonomous Vehicle Testing and Validation. Preprint on webpage at https://users.ece.cmu.edu/~koopman/pubs/koopman16_sae_autonomous_validation.pdf, 2016. 88

[95] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms.* Algorithms and Combinatorics. Springer Berlin Heidelberg, 2012. 134

[96] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. 122, 134

[97] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011. 26, 93

[98] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. 11

[99] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects Comput.*, 11(6):637–664, 1999. 9

[100] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, pages 593–604. ACM, 2017. 13

[101] A. Leitner, T. Ochs, L. Bulwahn, and D. Watzenig. Open Dependable Power Computing Platform for Automated Driving. In *Automated Driving: Safer and More Efficient Future Driving*, pages 353–367. Springer International Publishing, Cham, 2017. 86

[102] F. Leitner-Fischer. *Causality Checking of Safety-Critical Software and Systems.* PhD thesis, University of Konstanz, Germany, 2015. III, 4, 7, 10, 11, 15, 18, 22, 23, 24, 25, 26, 33, 34, 50, 52, 53, 54, 60, 61, 62, 64, 65, 71, 82, 159

[103] F. Leitner-Fischer and S. Leue. QuantUM: Quantitative Safety Analysis of UML Models. In *QAPL*, volume 57 of *EPTCS*, pages 16–30, 2011. 4, 7, 72

[104] F. Leitner-Fischer and S. Leue. Causality Checking for Complex System Models. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 248–267. Springer, 2013. III, V, 4, 7, 10, 11, 15, 22, 24, 25, 67

[105] F. Leitner-Fischer and S. Leue. Probabilistic Fault Tree Synthesis Using Causality Computation. *IJCCBS*, 4(2):119–143, 2013. 7, 22, 23, 26, 87

[106] F. Leitner-Fischer and S. Leue. SpinCause: a Tool for Causality Checking. In *SPIN*, pages 117–120. ACM, 2014. 10, 11, 49, 50

[107] S. Leue and M. T. Befrouei. Mining Sequential Patterns to Explain Concurrent Counterexamples. In *SPIN*, volume 7976 of *Lecture Notes in Computer Science*, pages 264–281. Springer, 2013. 9

[108] D. Lewis. *Counterfactuals*. Wiley-Blackwell, 2001. III, V, 10, 105, 114

[109] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Model-based Engineering in the Embedded Systems Domain: an Industrial Survey on the State-of-Practice. *Software and Systems Modeling*, 17(1):91–113, 2018. 4

[110] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer, 1992. 15

[111] H. Martin, K. Tschabuschnig, O. Bridal, and D. Watzenig. Functional Safety of Automated Driving Systems: Does ISO 26262 Meet the Challenges? In *Automated Driving: Safer and More Efficient Future Driving*, pages 387–416. Springer International Publishing, Cham, 2017. 86

[112] Microsoft Research. *The Z3 Theorem Prover*, 2019. https://github.com/Z3Prover/z3. 133

[113] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 29

[114] S. Neuhaus and A. Zeller. Isolating Cause-Effect Chains in Computer Systems. In *Software Engineering*, volume P-105 of *LNI*, pages 169–180. GI, 2007. 9

[115] Object Management Group. OMG Systems Modeling Language, Specification 1.5, 2017. `http://www.omg.org/spec/SysML`. III, V, 3, 87

[116] Object Management Group. Unified Modelling Language, Specification 2.5.1, 2017. `http://www.omg.org/spec/UML`. 3

[117] Object Management Group. UML Profile for MARTE, Specification 1.2, 2019. `https://www.omg.org/spec/MARTE`. 3

[118] P. E. O'Neil and E. J. O'Neil. A Fast Expected Time Algorithm for Boolean Matrix Multiplication and Transitive Closure. *Inf. Control.*, 22(2):132–138, 1973. 79

[119] opaal. opaal Test Folder. `http://opaal-modelchecker.com/opaal-ltsmin`, 2011. Accessed: 2018-11-08. 144

[120] D. C. Oppen. Complexity, Convexity and Combinations of Theories. *Theor. Comput. Sci.*, 12:291–302, 1980. 111

[121] D. B. Polsen and J. van Vliet. Concrete Delays for Symbolic Traces. Master's thesis, Department of Computer Science, Aalborg University, 2010. Available from https://projekter.aau.dk/projekter/files/32183338/report.pdf. 12

[122] A. Reynolds, V. Kuncak, C. Tinelli, C. Barrett, and M. Deters. Refutation-based synthesis in SMT. *Formal Methods in System Design*, Feb 2017. 13

[123] SAE. *J3016_201609: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, sep 2016. `https://www.sae.org/standards/content/j3016_201609`. 86

[124] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electron. Notes Theor. Comput. Sci.*, 55(3):357–369, 2001. 9

[125] T. Schmid, S. Schraufstetter, J. Fritzsch, D. Hellhake, G. Koelln, and S. Wagner. Formal Verification of a Fail-Operational Automotive Driving System. *CoRR*, abs/2101.07307, 2021. 12

[126] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller. Successful Use of Incremental BMC in the Automotive Industry. In *FMICS*, volume 9128 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2015. 12

[127] B. Schroeder, E. Pinheiro, and W. Weber. DRAM errors in the wild: a large-scale field study. *Commun. ACM*, 54(2):100–107, 2011. 96

[128] V. Schuppan and A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. In *INFINITY*, volume 149 of *Electronic Notes in Theoretical Computer Science*, pages 79–96. Elsevier, 2005. 11

[129] M. Sirjani, E. A. Lee, and E. Khamespanah. Model Checking Software in Cyberphysical Systems. In *COMPSAC*, pages 1017–1026. IEEE, 2020. 9

[130] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development - Technology, Engineering, Management*. Pitman, 2006. 3

[131] M. H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti. Formal Verification of an Automotive Scenario in Service-Oriented Computing. In *ICSE*, pages 613–622. ACM, 2008. 12

[132] V. Todorov, F. Boulanger, and S. Taha. Formal Verification of Automotive Embedded Software. In *FormaliSE@ICSE*, pages 84–87. ACM, 2018. 5

[133] UPPAAL. UPPAAL Benchmarks. `http://www.it.uu.se/research/group/darts/uppaal/benchmarks/#benchmarks`, 2017. Accessed: 2020-01-23. 123, 124, 126, 144

[134] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault Tree Handbook. *US Nuclear Regulatory Commission*, 1981. III, V, 4

[135] H. Wang, D. Zhong, T. Zhao, and F. Ren. Integrating Model Checking With SysML in Complex System Safety Analysis. *IEEE Access*, 7:16561–16571, 2019. 9

[136] S. Wang, A. Ayoub, B. Kim, G. Gößler, O. Sokolsky, and I. Lee. A Causality Analysis Framework for Component-Based Real-Time Systems. In *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 285–303. Springer, 2013. 12

[137] D. Watzenig and M. Horn. Introduction to Automated Driving. In *Automated Driving: Safer and More Efficient Future Driving*, pages 3–16. Springer International Publishing, Cham, 2017. 88

[138] J. Weiser. Derivation of a Minimal Representation of Incomplete Partial Orders from Event Sequences. Master's thesis, University of Konstanz, 2019. 77

[139] G. Weiss, P. Schleiss, C. Drabek, A. Ruiz, and A. Radermacher. Safe Adaptation for Reliable and Energy-Efficient E/E Architectures. In D. Watzenig and B. Brandstätter, editors, *Comprehensive Energy Management - Safe Adaptation, Predictive Control and Thermal Management*. Springer, 2018. 86

[140] L. Wu, K. Su, S. Cai, X. Zhang, C. Zhang, and S. Wang. An I/O Efficient Approach for Detecting All Accepting Cycles. *IEEE Trans. Software Eng.*, 41(8):730–744, 2015. 11

[141] W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint-Solving. In *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 243–258. Chapman & Hall, 1994. Full version of the paper is available from http://www.it.uu.se/research/group/darts/papers/texts/wpd-forte94-full.pdf. 12, 138

[142] S. Yovine. KRONOS: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997. 29, 105

[143] A. Zeller. Isolating Cause-Effect Chains From Computer Programs. In *SIGSOFT FSE*, pages 1–10. ACM, 2002. 9

[144] S. J. Zhang and Y. Liu. An Automatic Approach to Model Checking UML State Machines. In *SSIRI (Companion)*, pages 1–6. IEEE Computer Society, 2010. 9