Rhea Carlotta Rinaldo

Look both ways before crossing the street:

Combined Safety and Security Analysis for Autonomous Vehicles

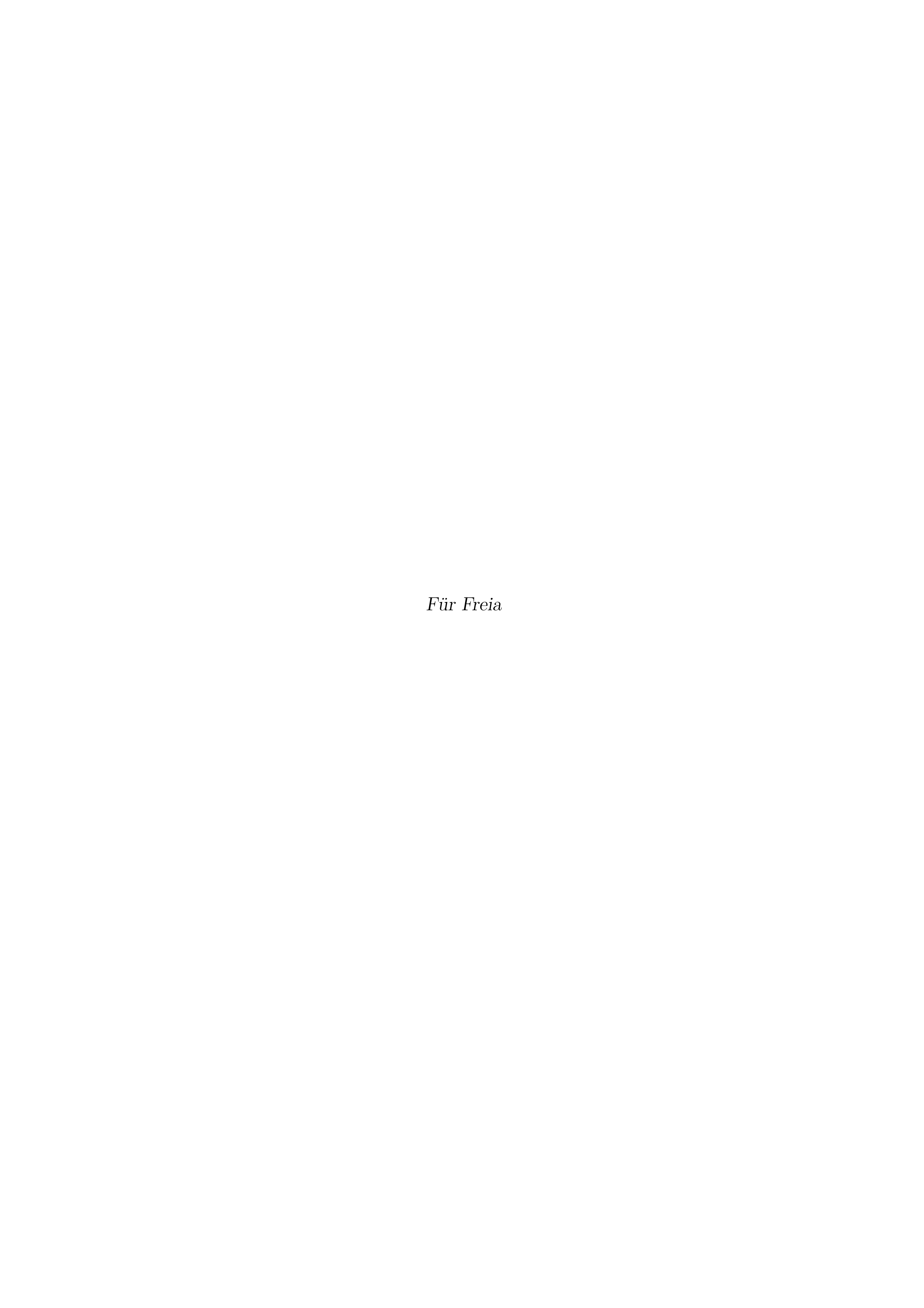
– Published Version –
Includes the addition of Section 9.3 that regards the application of the thesis' approach beyond the automotive domain.

Doctoral Thesis
First Examiner: Prof. Dr. Dieter Hutter
Second Examiner: Prof. Dr. Ina Schaefer
16th February 2024

Universität Bremen
Fachbereich 3
Informatik

Abstract

The advent of Autonomous Driving (AD) promises a revolution in transportation, with the vision of enhancing road safety, reducing traffic congestions and providing better accessibility. Though, with autonomous systems taking over vital functions previously performed by a human driver, ensuring their safety and security is paramount. Currently developed vehicles have not yet reached full autonomy, but already offer high driving automation. These modern vehicles are historically grown systems that consist of an orchestration of mechanical and electronic components with different criticality and maturity levels. High-level computing units running AI-based software are combined with classical actuators to make software commanded driving possible. Various interconnected devices are added to this structure for entertainment reasons and also to enhance the AD function through Vehicle-to-X communication. The result is a complex system consisting of components with critical functional dependencies, access relationships from the outside, redundancy specifications as well as repair and protection measures. Traditionally safety has always been a primary concern in automotive development. However, with increasing connectivity, security concerns are rising dramatically. While this risk has not remained unrecognized, security is still not targeted satisfactorily. Furthermore, it is still widely adopted to analyse safety and security separately in practice. This is a misjudged approach, given the fact that the two properties are intertwined: A security attack on a component endangers its correct operation and thus the system safety. Vice versa, a safety failure of a cryptography module increases the vulnerability of the components relying on it, favouring security attacks. The present dissertation addresses this issue with the development of a quantitative analysis method that is capable of modelling complex, critical systems and viewing the occurrence of safety failures and security attacks in parallel, as well as in dependence to one another. Therefore, a graph-based modelling of system level components and their dependencies is developed and a transformation into Continuous-Time Markov Chains formalized. In that, the occurrences of single failures and attacks of the individual components are modelled by state changes due to defined probability rates and their consequences to the system's capability of maintaining functional are reflected. The goal is to prepare for a quantitative analysis that yields the system failure probability over a specified time (observation period). The results can be used to support the development and the certification process of new vehicular architectures. In order to allow for a comfortable modelling and evaluation of complex system structures, this method is implemented in a tool called ERIS.



Für Freia

Contents

1 Introduction **8**
1.1 Motivation 10
1.2 Outline 12

2 Fundamentals **14**
2.1 Levels of Driving Automation 14
2.2 Modern Vehicle 16
2.2.1 Internal Communication Systems 18
2.2.2 Legacy Components 19
2.2.3 Self Driving System 20
2.2.4 Connectivity Components 22
2.2.5 Sensors 23
2.3 Safety and Security 25
2.3.1 Failure and Incident 26
2.3.2 Risk & Quality Assessment 27
2.4 Failure Behaviour and Tolerance Strategies 33
2.5 Assessment Paradigms 34
2.5.1 Probabilistic Model Checking 36
2.5.2 Stochastic Simulation: The Monte-Carlo Method 39

3 Modelling **42**
3.1 Component Criticality and Redundancy 43
3.2 Safety and Security 44
3.3 Fault Tolerance 47
3.4 Assessment Options 48

4 Formalization **50**
4.1 Dependency Graph 50
4.1.1 Application Example 56
4.2 Dependency Markov Chain 59
4.2.1 Transformation 60
4.2.2 Scalability 65
4.2.3 Evaluating the $\mathcal{DMC}$ 66

5 Modularization 70
5.1 Dependency Graph Division 72
5.2 Analysis 76
5.2.1 Choosing the Approach 77
5.2.2 Recursive Analytical Approach 78
5.2.3 Hybrid Approach 80
5.3 Heuristics 83
5.3.1 Accuracy 83
5.3.2 Well-defined Modules 86

6 Recovery 92
6.1 Recovery Approaches 93
6.1.1 Hardware Level 94
6.1.2 Application Level 94
6.1.3 System Level 95
6.2 Model Abstraction 96
6.3 Formalization 99
6.4 Application and Comparison 100

7 Automation: ERIS 103
7.1 Technical Insights 104
7.1.1 Menus 105
7.1.2 Toolbar 107
7.1.3 Toolboxes 108
7.1.4 Node Settings 108
7.2 Modelling with ERIS 109
7.2.1 Dependency Graph 109
7.2.2 Transformation 111
7.2.3 Recovery 115
7.2.4 Modularization 117
7.3 Evaluation 118
7.3.1 Performance 122
7.4 Example Analysis 123

8 Related Work 128
8.1 Markov-based Approach 129
8.2 AT-CARS 130

9 Conclusion 135
9.1 Summary 135
9.2 Challenges 137

9.3 Extended Application 137
9.4 Future Work 141

Author's Contributions **143**

Abbreviation **145**

List of Tables **148**

List of Figures **149**

Bibliography **151**

A PRISM Code Listings **163**
A.1 Modularization 163
A.1.1 Example 1: Problematic Security Modularization (Reach-path) 163
A.1.2 Example 2: Problematic Security Modularization (Reach-link) 166
A.2 Recovery 170
A.3 Automation 175
A.4 Related Work 184

B Evaluation Results **186**
B.1 Modularization 186
B.2 Recovery 189
B.3 Automation Analysis Results 190

Acknowledgements

I am grateful to my supervisor Prof. Dr. Dieter Hutter for offering invaluable guidance, on-point criticism and for convincing me to embark on this doctoral journey. Thank you for never getting tired of encouraging me to improve my very technical style of writing and turning it into a more comprehensive one – I hope it worked. I would also like to thank Prof. Dr. Ina Schaefer for accepting the position of the second examiner with true interest in my work. Furthermore, I would like to thank my former employer, the DFKI, for enabling this research opportunity and leaving me the necessary space for creativity, as well as my current employer, the IQZ, for supporting me in the final phase of this project.

Moreover, I would like to express my profound gratitude to my family and friends who have stood with me during this very special episode of my life. I am grateful to my mother, Freia Rinaldo, for her unconditional support and my brother, Vasco Rinaldo, who significantly formed my interest in computer science. Furthermore, I am particularly thankful to my father, Dr. Adam Zurek, who passed on some of his passion for research and science to me, though he would have preferred it to lie in the field of liberal arts. May he rest in peace. I am deeply appreciative for the constant support, countless technical discussions, and guidance, particularly in the phases of software development, from my partner Mattes Besuden.

Additionally, I would like to express my special thanks to Timo F. Horeis for being a great colleague and companion throughout several publications, conference visits, daily work and especially for imparting professional wisdom regarding reliability engineering. Thanks also to my former colleagues, especially Pascal Pieper and Christina Plump. Many thanks to Sina Hänsch for making life a little bit easier and Heino Kracke for his emotional support and for doing what he does best; provoking me in a loving, yet sometimes frustrating way. Further, thank you Lisa Jungmann for your help in making this thesis look more appealing. Lastly, I'd like to recognize my friend Ian Reeves for proof reading this dissertation on very short notice and Kieran Coyle for help with English grammar and wordings.

1 Introduction

Throughout the last decades an immense growth in the application of software and concomitant connectivity features of electronic systems could be registered. Systems that have previously consisted of mechanical components are now run by software controlled electronic components. In order to manage this software and keep it up to date, as well as for comfort and entertainment reasons, the connectivity of these systems has been increasing by the provision of, e.g., Wireless Local Area Network (WLAN) or Bluetooth interfaces. While this is a huge technological success and key to achieving higher degrees of automation up to full autonomy, it means significant challenges for safety and security: Former deterministic and relatively simple to proof systems are now rewritten in software, which is more prone to errors, difficult to assess in its completeness and potentially offers entry points for remote attacks.

The automotive vehicle makes for a prime example. While formerly being a purely mechanical system nowadays the vehicle consists of a plethora of mechanical and electronic components, employing various software applications and providing diverse communicating interfaces to the outside. By striving for higher degrees of automation in vehicles, we can witness this trend rising. Automation features have the primary goal of enhancing road safety. For instance, driver assistance systems are installed to provide the human driver with additional information and warnings, or even intervene in dangerous situations by, e.g., emergency braking. With 94% of car crashes being caused by human error according to a survey by the National Highway Traffic Safety Administration (NHTSA) published in 2008 [Adm08], the vision for driving completely autonomously is even greater: Manufacturers and advocates are claiming that with evolving technology autonomous driving, next to reducing traffic congestions and providing a better accessibility, will be capable of eliminating accidents caused by human error [She+21] entirely.

In order to achieve this high level of autonomy, current vehicle structures are being rethought: The centre piece of modern and future vehicles is becoming a system that combines manifold sensing components with exceedingly complex perception and decision making units coupled to classical actuators. Thereby diverse

sensors, prominently Radio Detection And Ranging (RADAR), Light Detection And Ranging (LIDAR) and camera, are being deployed in order to gather holistic information of the vehicle's surroundings. The fusion of the thereby sensed data is meant to create a perception that is equal or superior to human capabilities. High performance computing units running Artificial Intelligence (AI) based applications are required to analyse this data and determine decisions to manoeuvre the vehicle quickly and safely. In particular, objects and their velocity must be identified and trajectories to manoeuvre around them computed. Naturally this makes a highly critical and timely task in which potential failures or simply a slow computation are likely to have a fatal impact. To make this kind of computer commanded driving possible in general, these computing units must interact with the classical and critical actuators, such as the engine control unit and the brake and acceleration system, which have previously been operated mechanically by the human driver.

Furthermore, several interconnected devices are being installed. On the one hand, there are dedicated routers that support the autonomous driving task by enabling the communication with other vehicles or the infrastructure (Vehicle-to-X (V2X)), as well as the manufacturer's servers. According to the research of [She+21] a functioning V2X connection is mandatory to guarantee safety of completely autonomous vehicles. In addition to serving as a platform of exchanging information of the current driving situation, these interfaces are being used to provide software updates for various installed applications. On the other hand, there are comfort components like the frequently advancing infotainment system providing Universal Serial Bus (USB), WLAN or Bluetooth interfaces that also allow external devices like smartphones to connect with the system. This yields an overall system that combines highly diverse components with respect to their criticality, nature and maturity level of the operating software.

While it is questionable that this system is capable of eliminating all human-caused accidents, since the NHTSA also categorized causes such as "false assumption of other's action" and "decision error" [Adm08] which could equally be machine errors, with the exceeding use of software and the reliance on connectivity features, also new failure sources are being introduced. For instance, in October 2023 an accident with an General Motor's Cruise self-driving taxi occurred in San Francisco, hitting a person and dragging her along as a result of performing a pull-over manoeuvre. Even though the initial collision may have been unavoidable, Cruise lost their self-driving taxi license, as it was argued that it was not able to react appropriately to an accident and should have stopped with a person underneath it [Hei23]. Based on Cruise's statement "(...) the Cruise collision detection subsystem may cause the Cruise AV to attempt to pull over out of traffic instead of remaining

stationary when a pullover is not the desired post-collision response." [Cru23] it can be assumed, that this accident aggravated due to the system (supposedly) trying to reach the safe state. This incident could also be categorized as a decision error, which emphasizes the challenge of ensuring an autonomous vehicle's safety and security. Likewise, the demonstration of the vehicle's correctness to regulators and the public is becoming an increasingly difficult task.

1.1 Motivation

Within automotive development, safety has always been a primary concern. This fact is reflected by various safety-oriented developments, from driver assistance systems meant to increase road safety up to internals like safety-optimized communication systems such as the Controller Area Network (CAN) bus. With recent developments, we can see that safety concerns are even rising: An exceeding use of software-based components is becoming necessary to reach higher levels of automation, while these electronic components are being much more prone to errors than formerly deployed mechanical components [TD21] and it is impossible to write error-free software code [Tor00]. At the same time, the increasing connectivity and the partial reliance on it leads to a remarkable rise of security concerns. With security attacks essentially being the exploit of some open security gap, they are errors by design and thus have a high potential to scale; when a vulnerability is found it will most likely affect the entire vehicle series. Further, currently developed hybrid architectures combine unprotected legacy components with highly interconnected devices, creating room for fatal security attacks. As stated in [Wat19], there were about 50 million connected vehicles in the U.S. in 2019 already, and further, every major carmaker is integrating connectivity features in their vehicles. This trend and the recurring reports of security attacks on cars [Kos+10; MV15; Amm+20] clarify that security concerns must be treated with equal importance as safety.

Moreover, safety and security are partly intertwined: Security attacks can impact the system's safety, and vice versa, safety failures can condition the system's security. For example, the corruption of a critical component by an attacker endangers the correct operation of other components and the system. The other way around; the failure of a cryptography module increases the vulnerability of components that rely on it. Likewise, safety-oriented redundancy concepts must include the possibility of component outages and malfunction through successful attacks. In [Gla+15] it is argued that even though this problem has been recognized by the industry, challenges due to "the differing maturity levels, grey areas in law, [and]

dissimilarities in content“ exist, constraining an integrated evaluation of safety and security. As a consequence, it is still widely adopted to assess safety and security separately.

This thesis aims to address this issue by providing a methodology that is capable of handling these new vehicular architectures while viewing safety and security in combination. Therewith, this methodology is meant to support the automotive industry in proving the satisfaction of an automated and autonomous vehicle's safety and security requirements demanded by legislators and the public. The first step in achieving this is the development of a graph-based model that is capable of picturing the rather unique aspects of a critical, complex system such as the autonomous vehicle:

- the co-existence of components of varying criticality
- their failure behaviour and their vulnerability to security attacks in terms of probability rates
- redundancy specifications in combination with failure management and repair processes
- functional data dependencies
- commanding and connectivity dependencies through diverse interfaces
- as well as protection measures like the guarantees provided by cryptographic modules.

The resulting *dependency graph* offers a playground where safety and security events can happen in parallel but also in dependence to one another. In order to quantitatively analyse the operational capabilities of the modelled system a transformation from that graph into a Continuous-Time Markov Chain (CTMC) is determined. This allows to evaluate the system's operativeness in regard to a specified amount of passed time. In reality this could be, for example, the probability that the vehicle runs into a critical system failure during its envisaged lifetime. These analyses can, on the one side support the validation process of the vehicle, and on the other side support the development process by comparing the risk for critical system failures in different architectures.

1.2 Outline

The dissertation is structured as visualized in Table 1.1.

Table 1.1: Outline

Introduction and Scope	
Introduction	Motivation • Problem Statement
Modelling	Modern Vehicle • Automation Levels • Safety and Security • Risk and Quality Assessment • Assessment Approaches
Main Contributions	
Modelling	Modelling of Real World Effects
Formalization	Formal System Model as Dependency Graph • Transformation into Quantitative Model • Evaluation Principle • Scalability
Extensions	
Modularization	Dependency Graph Modularization for a Scalable Evaluation • Modularization Heuristics
Recovery	Component Recovery
Automation	Tool Support for an Automatic System Assessment
Closure	
Related Work	Delimitation to similar Methods and Approaches
Conclusion	Summary • Challenges • Extended Application • Future Work

The subsequent Chapter 2 provides an overview of the fundamental concepts required to understand the developed methodology. Thereby, the modern vehicle and its automation levels as well as the basics to risk and quality assessment are concerned. Furthermore, the principle and mathematical background of evaluation approaches applied in this thesis is given. Chapter 3 and 4 concern the cardinal modelling part of the approach. Thereby, Chapter 3 acts as a transition layer between the real-world effects to the formal methodology by discussing the predominant safety and security effects present in an (autonomous) vehicle that are meant to be regarded. Chapter 4 presents a graph-based formalization of this modelling and an accompanying solution in terms of a transformation into a Markov chain. Exemplarily, evaluation options and a statement regarding the scalability of the method are given, which shows its effectiveness, but also its Achilles heel, the exponential state growth. The subsequent Chapter 5 presents a modularization of the graph-based model with the goal of conquering the exponential state growth and making the methodology viable for analysing realistic architectures of autonomous vehicles. Thereby, an additional evaluation method that transforms

the approach into a hybrid model by making use of Monte-Carlo simulation and Markov analysis is introduced. Chapter 6 extends the existing concept by various possibilities of component recovery, in orientation to existing approaches from component and system level. In Chapter 7 the implementation of the methodology into a tool named ERIS is presented and an exhaustive example evaluation with it is performed. Lastly, Chapter 8 sets this work into relation of other similar approaches. The thesis is closed by a summary of the conducted work, a view on the challenges during its development as well as extended application possibilities and a discussion of relevant future extensions in Chapter 9.

2 Fundamentals

In this chapter fundamental aspects and methods which are used throughout this thesis are presented. The first Section 2.1 introduces the levels of driving automation and current pinnacle. This is followed by an overview of the basic structure and components of a modern, automated vehicle in Section 2.2, which is required to understand the later following application examples for the methodology developed in this thesis. The third Section 2.3 concerns the fields of functional safety and cybersecurity. Thereby, a general definition of either term is given and viewed in relation to the autonomous vehicle. Common parallels and differences between both fields are discussed and it is described how safety failures and security incidents are intertwined. Afterwards, a detailed summary of their risk and quality assessment is given and further divided into qualitative and quantitative approaches. This is followed by a brief review of the behaviour in failure cases and fault tolerance strategies. In Section 2.5 the different paradigms of assessment methods, namely *analytical*, *numerical* and *simulation-based*, are presented. It is emphasized that oftentimes approaches cannot clearly be classified, because they make use of varying types of techniques. The Section is closed by the introduction of two selective methods; probabilistic model checking on behalf of Markov chains and stochastic simulation with the focus on the Monte-Carlo method, which both find application the later chapters of this thesis.

2.1 Levels of Driving Automation

In dependence to price category and purpose, currently developed vehicles implement various types and different quality of automation features. Low-priced consumer vehicles usually only implement the legally required automation features, e.g., emergency brake assist (see also [Nor]). High-priced vehicles, however, tend to implement more sophisticated automation up to first implementations of autonomy in very particular use cases, for instance, autonomous parking assists or highway/autobahn lane keeping assists. With these assistance systems being capable of autonomously performing a defined task, their application is bound to very specific scenarios and at the moment they still require a human driver to be ready

to intervene in emergency cases, or if the scenario changes. Though, whilst still mostly part of research, also higher automated vehicles that can operate without a human control instance in specified scenarios are being developed. The different maturity of these implementations shows that a consistent definition becomes necessary when discussing and examining modern vehicles. Therefore, the Society of Automotive Engineers (SAE) has published an international standard (SAE J3016 [21c]) categorizing vehicles into six levels of automation. Figure 2.1 pictures these six levels of automation inspired by the visualisation made in [SAE21]. The first

Level 0	Level 1	Level 2	Level 3	Level 4	Level 5
No Driving Automation	Driver Assistance	Partial Driving Automation	Conditional Driving Automation	High Driving Automation	Full Driving Automation
A human is driving whenever these support features are engaged.			A human is not driving whenever these automated driving features are engaged.		
The support feautres must constantly be supervised by a human driver.			A human must take over on request.	These automated driving features do not require human driving/supervision.	

Figure 2.1: Levels of Driving Automation by SAE J3016 [21c; SAE21]

three levels refer to solutions that strictly require a human driver and are thus often called automated, while the other levels are referred to as autonomous. In level 0 the human driver is completely in control, however, safety mechanisms like the Anti-lock Braking System (ABS) may exist. In level 1 assistance systems specific to the Operational Design Domain (ODD) are present, supporting the human operator in the driving task. Thus this level concerns systems as the Lane Departure Warning System (LDWS) which is meant to work in control with a human driver. Level 2 describes systems with an active automation that is taking over vital parts of the driving task. However, the human operator must supervise this task and also perform several subtasks himself. For instance, advancements of the LDWS that employ automatic steering and acceleration that must be monitored and supervised by the human operator at any time. In level 3 an ODD-specific performance of autonomous driving systems is active, which does not require explicit supervision, yet the human operator must be ready to take over whenever the system requests it. An Automated Lane Keeping System (ALKS), which can be seen as a further advancement of the LDWS by not only warning the driver but

steering and holding the lane autonomously in a defined ODD, matches this category. In that way, driving on the highway can be performed autonomously under the condition that the human takes over whenever the vehicle is leaving that ODD, which may be due to a planned road change, but also due to the experience of unplanned scenarios like accidents. Level 4 defines systems that cannot only drive autonomously in specified ODDs, but also engage autonomous fallback systems. Here the present human is merely a passenger. Lastly, level 5 describes systems that unconditionally (not ODD-specific) perform the entire driving task and fallback system autonomously. It should be noted that the level of an automotive application is not only describing its capabilities, but also the safety requirements it must fulfil. Following from that, in level 2 and to some extent in level 3 systems the safety responsibility can be transferred to the human operator, simplifying the proof of requirement satisfaction to the regulators. Likewise we can see that it is significantly easier to prove the satisfaction of the safety requirements of level 4 applications than of level 5, as they *only* have to function in a very particular environment. Thereby, despite the vision of autonomous vehicles being to become safer than the human driver, the minimum condition is to be as safe as an attentive and skilled human driver (e.g. seen in the safety performance models of ALKS given in the UNECE 165 regulation [21a])

The leading manufacturers of autonomous vehicles, Waymo[1], Cruise[2] and Argo AI[3] together with Ford and Volkswagen Group, are currently developing and testing level 4 vehicles [SS21]. Additionally to that, several other automakers are working on autonomous driving systems or providing lower level autopilots. For example, Tesla[4] is providing an autopilot that can be categorized as level 2 with a tendency to level 3 [SS21; Per22] and is additionally working on fully autonomous vehicles [Gon22]. Traditional manufacturers like Mercedes-Benz[5] and BMW[6] reach automation level 3 with their self-driving technologies [Per22]. In this thesis the focus is set on vehicles with level 3 or higher.

2.2 Modern Vehicle

The modern vehicle is a historically grown system that consists of an orchestration of mechanical and electronic components. Thereby, electronic components

[1] https://waymo.com/ visited on 22nd January 2023
[2] https://getcruise.com/ visited on 22nd January 2023
[3] https://www.argo.ai/ visited on 22nd January 2023
[4] https://www.tesla.com/ visited on 22nd January 2023
[5] https://www.mercedes-benz.de/ visited on 22nd January 2023
[6] https://www.bmw.de/ visited on 22nd January 2023

are equipped with dedicated firm- and software to manage and control the behaviour of the mechanical parts (e.g the ABS controlling the brakes). By striving for higher levels of automation, electronic components are in duty of taking over more and more vital controlling tasks that have previously been performed by the human driver. Depending on the level of automation, several assistance system up to an entire Self Driving System (SDS) are implemented in the vehicle. As a consequence, the implemented software is becoming increasingly complex. Accompanying, powerful perception mechanisms become mandatory and make substantial sensor usage necessary. This leaves the modern vehicle a complex system that integrates a plethora of components with varying properties and criticality levels. Precisely, highly safety-critical components that guarantee the driving functionality of the vehicle, such as the engine control unit, are combined with interconnected comfort devices, like the infotainment system and highly sophisticated and intelligent computing units that perform autonomous driving tasks (AI computers). Summarizing, a hybrid consisting of normal, historically grown vehicle components, referred to as *legacy components*, and the newly added AI and sensing components arises.

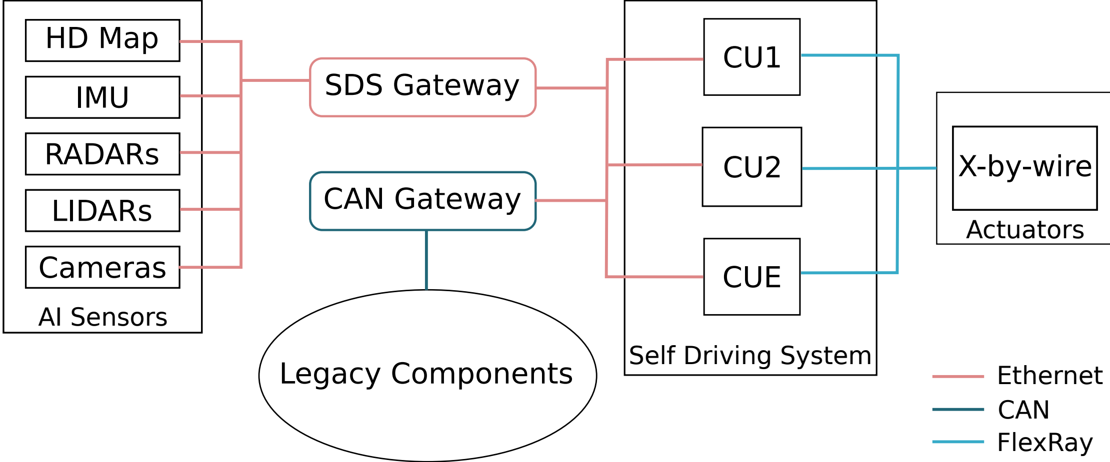


Figure 2.2: Modern Vehicle Architecture with SDS

While the architecture of farther future autonomous vehicle may be reinvented to follow a more centralized approach, in the course of this thesis such a hybrid system is assumed. For manifold reasons, it is additionally conceivable that also near future autonomous vehicles will maintain this architecture: First of all, the legacy components have been deployed for decades and have been tested heavily to ensure functional safety. Secondly, on-going contracts between vehicle and component manufacturers exist, making it unlikely that these components will be completely erased from the vehicle in the near future. And lastly, development,

testing and regulation of new vehicle parts and applications are tedious and lengthy processes.

Figure 2.2 pictures the assumed architecture of a modern vehicle previously established for a case study of the methodology presented in this thesis [RHK21]. On the left hand side, sensing components that provide the necessary data for the AI perception are sketched. These are tethered to the computing units of the SDS via Ethernet and by making use of a dedicated gateway. The legacy components are internally placed on several, gateway-connected CAN buses. To communicate with the other vehicle components (e.g. the SDS) CAN messages are transformed to Ethernet packets and communicated via the CAN gateway and vice versa. The SDS has a FlexRay connection to the (mechanical) actuators that perform x-by-wire tasks, namely drive-by-wire, steer-by-wire and break-by-wire. This connection is essential to establish the driving functionality based on the processed data of the legacy components and the AI perception mechanisms. To provide a more detailed insight into these different vehicle components of current and near future autonomous vehicles, the following gives an introduction to the most important ones.

2.2.1 Internal Communication Systems

In order to meet the requirements of the divergent components of a modern vehicle, different communication systems for commanding and exchanging data between them have been developed. Table 2.1 gives an overview of the most commonly implemented systems and technologies (see [WWP04] for legacy bus systems and [Röd17] for Ethernet). Local Interconnect Network (LIN), CAN and FlexRay are

Table 2.1: Vehicular Communication Systems

Type	Application	Bandwidth
LIN	Low-level communication systems	20 kBit/s
CAN	Soft real-time systems	1 MBit/s
FlexRay	Hard real-time systems (X-by-wire)	10 MBit/s
MOST	Multimedia, Telematics	24 MBit/s
Ethernet	Smart Systems, Multimedia	10 MBit/s – 1 GBit/s

used for low-level communication networks of often safety critical components. Thereby, LIN is the cheapest with the lowest data rate and often used to connect simple sensors, such as the rain detection sensor, to other bus systems. CAN and FlexRay are more developed and highly safety optimized, whereas FlexRay provides a higher data rate and two channel redundancy. Both are not well suited

for multi-media transmissions, as they are too cost-intensive (financially) and their rather low data rate is not befitting real-time video and audio transmissions. Media Oriented Systems Transport (MOST), on the other hand, was especially designed for this task and provides, amongst other things, a very a high data rate. Automotive Ethernet is seen as the communication system for future vehicles, because it makes a cost-efficient solution for providing a very high bandwidth and being real-time capable. This is key to enabling automation features, since the data of the various perception sensors is fused together and requires a high bandwidth to be transmitted to the processing unit (SDS). Further, its timely processing and the resulting commanding of the actuators is essential for a safe autonomous driving process. Furthermore, security requirements are rising with the increasing connectivity of vehicles and the legacy bus systems are often mostly unprotected against security attacks (see [WWP04; MV14]). Ethernet on the contrary, provides several possibilities for ensuring security and privacy means. Tested and proven solutions already exist in the non-automotive context, which may be transferred to the usage in automotive Ethernet. In the farther future it is conceivable that most of the commonly used bus systems will be replaced by automotive Ethernet.

2.2.2 Legacy Components

Legacy components are seen as the historically grown components of a normal vehicle. These are usually based on some Electronic Control Unit (ECU) that runs a dedicated firmware, like the engine control unit or the body control module. As mentioned earlier, in the future it is likely that many currently implemented legacy components will be replaced by software processes of the SDS. However, for current and near future vehicles, several legacy components persist. Depending on their task and criticality, they are situated on the different vehicle buses as introduced before. Typically there is a MOST bus connecting multi media components such as the navigation system, telematic control unit, the radio, the infotainment system and so on. Then there exist several different CAN buses. For instance, a comfort CAN bus that contains components like the energy management, body control module, parking assistance, door controls etc., and the highly critical motor CAN bus containing, among others, the engine control unit, the ABS, the airbag system. All of these buses are connected to the On-Board Diagnostic II (OBD-II) interface for diagnosis purposes. This layout highly depends on the vehicle make and model, however, as an example [Pen09] depicts some actual bus topologies and their components.

2.2.3 Self Driving System

The SDS is the centre piece of an autonomous vehicle that takes over the task of the human driver. It consists of a number of high performance computing units running an array of software applications. These applications can be grouped into six functional modules: localization, detection, prediction, mission planning, motion planning and actuation, following the software stack proposed by Autoware [Kat+18]. This open-source Robot Operating System (ROS) based framework for autonomous driving is visualized in Figure 2.3. Due to the high safety

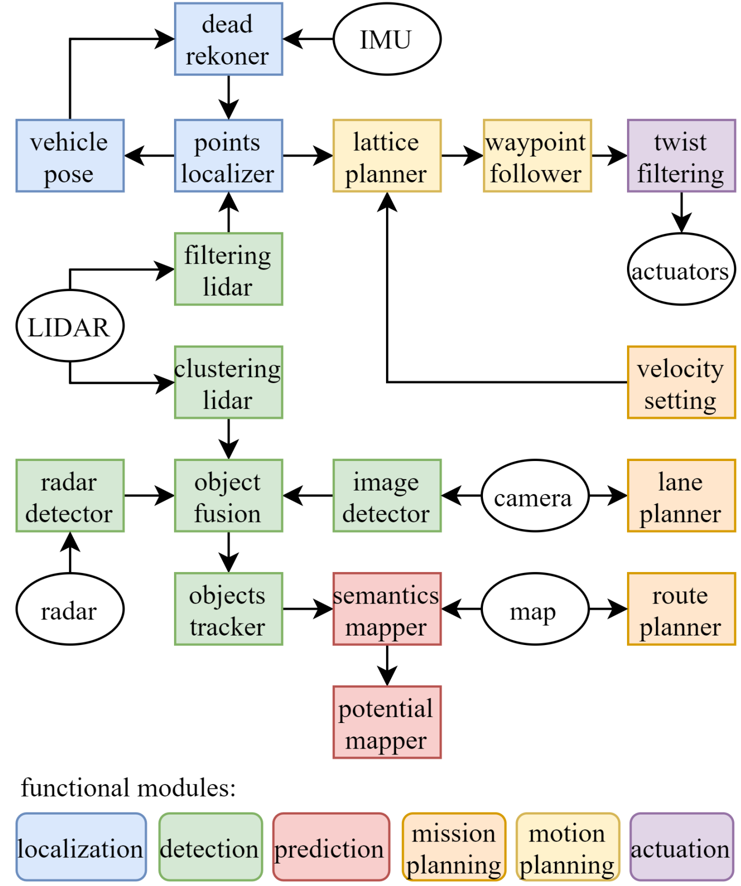


Figure 2.3: Autoware Software Stack [Kat+18]

criticality of the SDS its computing units are usually redundantly designed. The SDS pictured in Figure 2.2 shows three computing units, with $CU1$ and $CU2$ being completely redundant and CUE depicting an emergency computing unit which is acting as fallback system. Several different redundancy concepts are plausible with this structure. Recently, there has been an increasing discussion

on the implementation of Fault Detection Isolation and Recovery (FDIR) processes, formerly applied in the space domain, in vehicles (see for example [Kai+20] and Section 6.1.3). Roughly speaking, these processes allow for the detection and identification of faulty hard- and software. Once the affected part is identified, it is isolated and a recovery process is triggered to maintain a safe system state. Thereby, the availability of the system can be drastically increased. On behalf of this, the vehicle gains the possibility to adapt itself. First efforts regarding this have been made in the SafeAdapt Project[7] [SDW14] or more recently in SelfAutoDOC[8].

A sophisticated way to achieve this self-adaption mechanisms is to use the two redundant CUs in the following way: The software applications between those CUs are split somewhat equally, but the respective other CU is running an emergency copy. Therewith, each CU is potentially capable of performing the entire driving process. This requires to consider two different software process states: active and active-hot. Thereby the active state describes the normal application operation, while active-hot describes some sort of stand-by mode, where the application is performing its designated task but not communicating to other applications and the actuators. Therewith, active-hot applications can run a degraded mode. The idea with it is that if one CU fails to provide its functionality or to keep the application alive (due to an overload or the application itself being erroneous), the active-hot instances running on the other CU can be switched to active, taking over the vital functions previously performed by their copy (see also [RHK21]). This means that in normal operation the two computing units work in parallel, however, minor problems up to the entire outage of a CU can be covered by the respective other CU to maintain a safe operational state. Additionally to that, the emergency computing unit CUE is installed in an analogous way that takes over the control in case both, $CU1$ and $CU2$ fail, to ensure that at least manoeuvring and halting in a safe position (e.g. on the verge) can be performed safely. Therefore, CUE is running a degraded version of the most necessary applications. However, a takeover of CUE does not necessarily mean that both CUs have failed hardware-based, it could also mean that the same software application on either device ceased to operate correctly. As a detail, therefore it can be highly beneficial to run a differently programmed application on at least the emergency computing unit.

[7] `https://www.safeadapt.eu/` visited on 30th September 2023

[8] `https://www.tttech.com/innovation/research-projects/other/selfautodoc` visited on 30th September 2023

2.2.4 Connectivity Components

Connectivity components describe the type of components that offer interfaces to communicate with the environment outside the vehicle. Commonly applied technologies are Bluetooth, Global System for Mobile Communications (GSM) and WLAN, though some components also offer physical interfaces like USB. While these connections are very useful and even become necessary at high automation levels [She+21], they also can open vulnerabilities by acting as initial entry points for potential attackers.

The telematics unit provides an interface to the backend and external servers of the vehicle's manufacturer or maintainer. This connection is used to provide the vehicle with various information, such as traffic data and weather conditions. But it is also used to perform software updates, (re)-configurations and trigger repair behaviour of different vehicle components. Further, it can be used to provide the servers with internal housekeeping data for, e.g., diagnostic purposes. This makes the telematics unit a key component of maintaining the functionality of the modern vehicle during its envisaged lifetime. Additionally to that, a Vehicular ad hoc Network (VANET) router to enable the communication with other autonomous vehicles, or smart infrastructures, is connected to the computing units of the SDS. While this communication system is still experimental, the idea is that it can be used in the future to share various information on the road with other vehicles and smart infrastructures. This information shared among vehicles includes current driving speed, braking behaviour but also more general information such as traffic and weather conditions. Further, infrastructures, naturally traffic lights, but also parking lots are integrated in this system to signal their status information including the amount of currently available parking spaces (for more information of the VANET technology and applications view [LA21]). While VANET communication offers great solutions in supplying diverse information, it also is very risky from a security point of view, as false data may easily be spread and the existing interfaces misused. Though this risk has not remained unrecognized and several research publications targeting this problem exist, e.g. [Has+17].

The infotainment system can be seen as a connected component as it usually offers connection interfaces to multimedia devices via both, wireless and wired interfaces. Its function volume depends on the concerned vehicle and manufacturer, however, it is usually characterized by being connected to the CAN bus and providing some Human Machine Interface (HMI) to the passengers. In this way, the passengers can steer functions, in particular, the climate control, the lights, the radio and other, which makes a commanding connection to other CAN bus components ne-

cessary. State information such as the heat and the adjusted settings, but also the current driving speed are displayed to the passengers or used for further systems (e.g. automatically adjusted radio sound in relation to driving speed). The OBD-II interface provides a direct, physical connection to the CAN bus which can be accessed from inside the vehicle. The interface is used for diagnostic purposes and thus usually only accessed in the repair shop with special devices, though applications and tools for passenger use exist as well. Usually the OBD-II interface is write-protected, requiring the authentication of the device in order to send commands, while the traffic may be read by anyone. As stated by Ammar et al. [Amm+20], the ISO 15031-7 [13] defines the access control of the interface and suggests a simple challenge-response protocol with pre-shared keys and a dedicated crypto function among the connected device and the vehicle. However, this procedure is only very vaguely described which in practice leads to unprotected OBD-II interfaces or poorly implemented (non-standardized) protocols with improper key length [Amm+20].

Several ways of tempering with these connectivity components are conceivable: existing interfaces or connected devices such as smartphones and USB sticks may be misused in masquerade attacks where the attacker impersonates authorities, for one, the manufacturer's servers to deploy malicious commands or malware. As mentioned, the poor authentication scheme of the OBD-II interface can be attacked by replay or brute force attacks, yielding direct access to the CAN bus allowing the attackers to clone vehicle keys or insert other malicious commands. For more information, Miller and Valasek provide an excellent overview on remote attack surfaces in vehicles [MV14]. Usual countermeasures to lower the risk for security attacks are: Following cryptographic guidelines, keeping the used communication protocols up to date and requiring authenticated and/or encrypted communication. Lately there has been an increasing research regarding automotive specific security, for instance, domain specific Intrusion Detection Systems (IDSs) (see for example [Al-+19]). and installing firewalls and IDSs.

2.2.5 Sensors

Modern vehicles are equipped with an increasing number of sensors, each fulfilling different purposes. Some are meant to enhance safety, such as tire pressure sensors and temperature sensors. Others enhance the comfort of the human driver, like the rain detection sensor which yields the necessary data to automatically activate the wipers. In a normal vehicle with a human control instance these sensors are mostly uncritical, however, for vehicles with higher degrees of automation these

sensors become essential to perceive the status of the vehicle and its environment.

In accordance to the increasing levels of automation, the number and the significance of these sensors is rising. For the environment perception commonly RADARs, LIDARs and different types of cameras, e.g., 3D cameras are used to deliver the necessary data. The data produced by these imaging devices is combined and enriched with further information, specifically positioning and localization based on HD maps and Global Positioning System (GPS) or Global Navigation Satellite System (GNSS) (sensor fusion). Based on this preprocessed data, the applications of the SDS are executed.

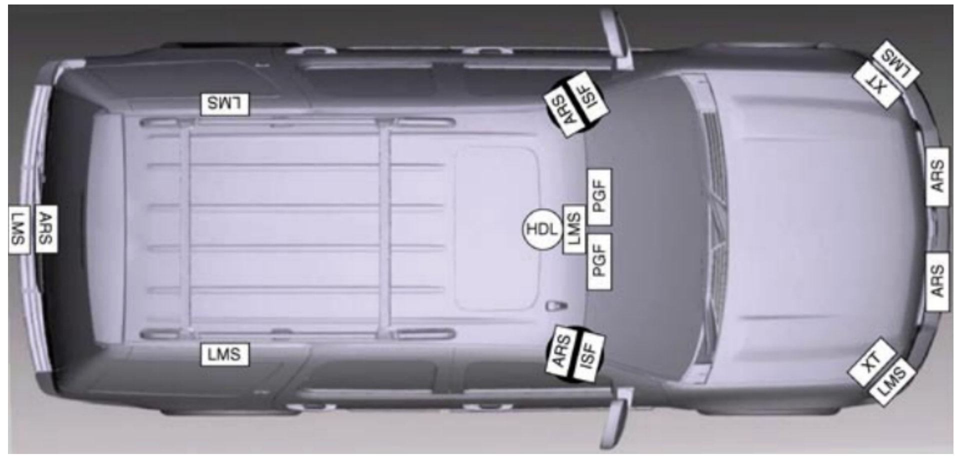


Figure 2.4: Boss Sensor Placement [UA+09]

To provide an example setup of these SDS perception sensors, Figure 2.4 shows the sensor structure of the autonomous vehicle BOSS, developed by the Carnegie Mellon University for the DARPA Urban Challenge [UA+09]. The vehicle is equipped with multiple RADARs and LIDARs that cover different ranges:

- long-range RADAR (ARS)
- long range LIDAR (XT)
- mid-range LIDAR (ISF)
- low-range LIDAR (LMS)

Additionally to that, a low-range 360° LIDAR is placed on top of the vehicle (HDL) and the only sensor in this setup that is capable of providing 3D information. Two high dynamic-range cameras (PGFs) used for road estimation. Furthermore, a GPS sensor is used to support the localization in the lane. As mentioned in

[DRU08] the redundant design enables tolerance to single sensor failures. Due to the critical consequences of false perceptions, a redundant and fault tolerant sensor structure is highly relevant.

2.3 Safety and Security

The concepts of functional safety and cybersecurity are complementary, yet also share strong parallels. Consider a system that is operating in some environment with the system being able to affect the environment and vice versa. Safety can be described as the inability of the system to affect its environment in an undesirable way and security as the inability of the environment to affect the system in an undesirable way (see [Lin+06]). In regard of automated and autonomous vehicles functional safety relates to the protection of the vehicle's passengers, other road users, and the surroundings against the consequences of internal vehicle failures. Security, on the other hand, relates to the protection of the vehicle's safety, including its passengers, road users and surroundings, against the consequences of malicious attacks originating from attackers that reside in the environment. Furthermore, it concerns the protection of private information, hence privacy, e.g., concerning the passengers private data and ensuring that it is not leaked to the environment, for instance, to insurance companies without consent.

Albeit safety and security are complementary concepts their effects and their related measures share strong parallels and cause interdependencies. Safety failures can increase the security-wise vulnerability of the system, if, as an illustration, security enhancing components like cryptographic modules or firewalls are affected. While some security incidents such as attacks on the privacy of the user do not affect the system's safety at all, Denial of Service (DOS) or ransomware attacks on system components can lead to critical consequences for the safety of the system, in dependence to the criticality of the affected component. In some cases safety and security solutions have very similar goals. For example in messaging, Error Correcting Codes (ECCs) (safety) and Message Authentication Codes (MACs) (security) both aim to ensure the integrity of the communicated message. Though, ECCs address the correction of bit flips caused by noisy communication channels and MACs focus on the protection against malicious tempering of messages like masquerade attacks.

However, a valuable safety measure can create a security gap at the same time. The automotive CAN bus illustrates this excellently. As we saw earlier, the CAN bus is highly safety-optimized to ensure the correct and timely communication between

critical vehicle components. Though, security concerns were not regarded during its development and consequently it contains several safety enhancing features that can greatly play into the attacker's hands. For example, CAN bus controllers (the sending and receiving unit) are switched into an *bus-off* error state which prohibits them from sending any messages to stop a faulty controller from causing trouble. Simplified speaking, this state is triggered after a certain threshold of erroneous messages sent and received by the concerned controller is detected. Work like [FS17] have shown how this initial safety measure can be exploited by an attacker to evoke the bus-off state of a certain component and thereby actually violate the system's safety. Attacks on CAN bus controllers could be avoided by the implementation of security measures. [NR16] for example proposes an authentication scheme for CAN bus messages that provides can bus controllers with the ability of verifying the authenticity of received messages and also monitoring the bus to detect fraudulent messages, allegedly sent by itself.

So if we have a solution, why is it not a common practice yet? The problem is that the addition of security measures can also pose strong implications to safety: An increased computational power on both the sending and the receiving controller is required to compute and verify cryptographic hashes, and the message length increases accordingly. Following from that, the required time until a message or command is actually processed by the intended receiver heavily increases. For highly critical commands, like braking, authentication can seriously lower security risks, however, due to their strong time-requirements safety risks may rise simultaneously.

2.3.1 Failure and Incident

Safety problems are usually distinguished by the three categories: Error, fault and failure (note that further distinctions exist but are not relevant for our scope). While a safety failure is mostly consistently defined as the inability of the component or system to perform its designed function (see [Hor+22]), various different and partially contradicting definitions for error and fault circulate. For this thesis, the succeeding definition, previously published in [Hor+22], is followed. An error is defined as the discrepancy between the measured, computed or received value and the intended value, which ultimately may lead to a failure. Faults, on the other hand, are defects of the system or component, for example, a defective memory section. Thus, errors are consequences of faults, however, a fault must not lead to an error, for instance, if the defective memory section is never used or faulty code never executed.

Akin to this distinction, in security it can be separated between vulnerability, exploit, incident and attack. A security vulnerability is the flaw or weakness of a component such as a design-flaw of delicate software, e.g., of the implemented communication protocol, or the absence of proper protection mechanisms, which could be taken advantage of by an adversary (see also [Cyb]). An exploit translates to a sequence of actions that can be taken to abuse a specific vulnerability such as (malicious) program code. On successful execution, the exploit leads to a security incident; the compromise of the viewed system or data which may propagate to further damage outside the system [Lin+06]. Thereby an incident summarizes both, intentional and unintentional exploits. Unintentional exploits are, for example, accidental access to restricted areas by abusing a vulnerability in access control, or the opening of phishing E-mails that leads to malicious code injections. Intentional incidents are attacks, such as brute-force attacks on credentials of someone else to access restricted areas and so on. Here is where the interdependencies of safety and security have their root: A security incident or attack may lead to a safety failure for several reasons, including but not limited to:

- Malicious tempering with input data (falsification) that leads to incorrect output behaviour, in other words errors.
- DOS or ransomware attacks that lead to a (temporary) irresponsiveness of the component (failures).
- Entire take over of the component by malware distribution that leads to undefined behaviour including the above.

Furthermore, a safety error can simultaneously be a security vulnerability. As a demonstration, automotive RADARs are prone to interference, hence the modification or disruption of a RADAR signal due to other signals [Yeh+17]. Interference can be both intentional (security attack) if an attacker actively sends radio signals to interfere with a specific RADAR, or unintentional (safety error) if the signals result from other automotive RADARs sending on the same frequencies. In either way, interference may lead to failures of the subsequent processing functions, such as object detection, which potentially can result in fatal consequences.

2.3.2 Risk & Quality Assessment

Various approaches for assessing the risk and quality of the safety and the security of a system exist. While qualitative approaches aim to subjectively rank the risks of scenarios in a system in relation to the risk of other scenarios, quantitative approaches aim to measure it numerically in regard to a predefined scale [Alt95]. As

stated by Zio [Zio07], qualitative risk yields from damage plus uncertainty, while quantitative risk is basically the product of the damaging consequence and its probability of occurrence. Qualitative approaches often make use of semi-quantitative, simplified scales such as *low, medium and high* to determine the risk of a safety hazard, i.e. an event or effect that can lead to an undesirable effect for the health or life of the asset, or a security threat, an event or effect that has undesirable consequences for the information in the system (see also [Lin+06]). The automotive functional safety standard ISO 26262 [18] defines a qualitative risk assessment that essentially consists of three steps: item definition, Hazard Analysis and Risk Assessment (HARA), and determination of a functional safety concept. In the first step the concerned item (component or system that satisfies a vehicle function) is defined by viewing its functional behaviour, legal requirements, constraints like functional dependencies etc. In the next step, hazards are identified and classified with respect to their consequences, resulting in an Automotive Safety Integrity Level (ASIL) rating (an adaption of the prominent Safety Integrity Level (SIL) classification for the automotive industry) that defines the risk of the system in terms of a semi-quantitative scale regarding severity, exposure and controllability. The ASIL determination is essential for proving the safety of the system to regulators and the public. Safety goals, which determine how to cope with the identified hazards, are defined and verified. Based on that, the functional safety concept is built, specifying requirements that are needed to satisfy the identified safety goals. A similar approach is defined for security in the ISO/SAE 21434 [21b], which can be analogously performed by a Threat Analysis and Risk Assessment (TARA) and, e.g., by making use of the STRIDE[9] threat model. Thereby, instead of viewing items, *assets* are concerned which are *something* for which a compromise of its security properties may lead to the damage of an item. In the first step, these assets and their damage scenarios are identified. Afterwards, for each damage scenario one or more threat scenarios are identified. These threat scenarios describe malicious events that can lead to the occurrence of the damage scenario. The attack paths that can lead to the threat scenarios are analysed and a rating of their feasibility is determined by qualitative metric, e.g., an Common Vulnerability Scoring System (CVSS)-based approach. Parallelly the impacts of each threat are rated regarding their severity. The results of both ratings constitute to the determination of risk values; a value between 1 and 5 specifying the risk per threat scenario. In the end, the established information is used to define treatment decisions for each threat. As an particular example, *reducing the risk* through the addition of cryptographic features.

[9] by Microsoft https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN visited on 17th July 2022

The essential steps performed in either risk assessment are comprised in Figure 2.5. Quantitative approaches can be used to complement these qualitative risk

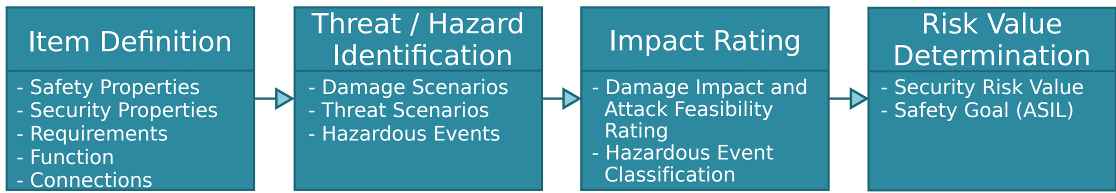


Figure 2.5: Essential Steps of a Safety/Security Risk Assessment (simplified)

assessments, by providing a risk evaluation on sublevels to support the rating process or to verify the obtained safety and security goals. For example, the ISO 26262 [18] states that quantitative approaches should be used to assess the risk of hardware failures. Furthermore, they can be used as an independent, parallel analysis, which can be especially beneficial in highly complex systems where qualitative approaches are difficult to apply.

Quantitative approaches are usually targeted on calculating the entry of an undesired event, such as the effect of the threat or hazard, in terms of a probability or occurrence rate. As an example, the probability that a system or a component fails safety-wise (failure probability) or is successfully occupied by an attacker (attack probability). Since security attacks usually have indirect consequences to the safety it is difficult and sometimes not useful to distinguish between them.

The main difference between a safety risk and a security risk is their behaviour over time. Safety risk is usually assumed to behave constantly over time, meaning that a component's remaining life time does not depend on its already progressed lifetime [MP10] and thus can be described by an exponential distribution.[10] However, in dependence of the viewed system, additional factors, in particular, ageing and wear of mechanical components slowly contribute to component failure. Given that, the failure rate is often not actually constant but changing over the deployment time. Yet, since a constant failure rate immensely simplifies safety and reliability computations an exponential distribution is usually considered any way for electronic components [MP10]. A security risk, on the contrary, behaves dynamically, not necessarily increasing over time but changing in dependence to the occurrence of certain events (exploits). The risk for a security threat to happen may be constant in the beginning, but increases heavily in the moment a new vulnerability is found and decreases equally fast as soon as the security gap causing

[10] A probability distribution of the time between continuously and independently occurring events at a constant average rate in a process.

the vulnerability is closed. However, in general one could assume that an ageing or rather time-dependent effect also exists for security:

- Implemented cryptographic measures may get outdated by becoming reversible with the increasing technological progress and the associated increase of computational power. For example, certain (small) groups used in the Diffie-Hellman key exchange protocol as suggested in [Adr+15].
- Applied cryptographic keys lose their freshness and may get exposed or reverse engineered by cryptanalytic attacks.
- The attacker's behaviour constitutes of different phases; the learning phase, the standard attack phase and the innovative attack phase according to [Mad+02]. In both the learning and the innovative phase, the probability for successful attacks is much lower. Thus, it can be derived that also the probability for an attack is increasing in relation to the experience an attacker is gaining.

Though, the mere existence of a vulnerability is not the only factor, as it may never be exploited. This requires an active attacker to do so which makes it valid to take other factors such as the attack vector and the attacker's motivation into account. Nevertheless, also security risk is often validly described by an exponential distribution. As stated in [Mad+02], analysis of collected data has shown that the time between security breaches in the standard attack phase, the phase with the highest probability for success, is exponentially distributed.

Quantitative approaches excel in application fields where a precise determination of hazards and threats is difficult or impractical, yet a failure or attack behaviour can be observed or estimated and modelled accordingly. For safety, Fault Tree Analysis (FTA) is a historically grown method that can be extended in the base form to allow the calculation of failure probabilities. Attack Trees (or Graphs respectively) [Sch99] can be seen as the equivalent method for security, where threats and possible attack scenarios are described and annotated with occurrence probabilities to estimate the overall attack probability. While being perhaps the most pervasive method for safety, Monte Carlo Simulation (MCS) is also often used in the security field. Thereby, usually the system's failure or attack behaviour is modelled and the future behaviour predicted, which is explained in more detail in the upcoming Section 2.5.2. Further, Markov-based approaches are largely applied in either field as it can be seen in [Mun+15] and [ASB07]. They induce a state-based modelling that is usually used to reflect different system states. Changes between these states are performed by transitions that occur on a defined rate or probability. In the end, the probability for reaching a specific state or

group of states, that relates to a certain system state, for instance, *failed* can be evaluated.

In addition to attack and failure probabilities, concomitant quality attributes of safety and security can be assessed. The key attributes that lead to a safer system are reliability, availability and maintainability. Often further properties like resilience and survivability are discussed, which are however, based on these key attributes. Similar attributes for security exist, such as confidentiality, integrity and availability (the CIA triad), as well as authenticity, non-repudiation and privacy. Though, these attributes mainly focus on the level of information security, while in risk assessment we usually target the system level and are interested in the implications of security incidents to the system's safety. Therefore, for security we are also interested in the fulfilment of safety objectives (which can include attributes like confidentiality, authenticity etc.).

Satisfying these safety and security attributes contributes to a dependable system; a system such that "reliance can justifiably be placed on the service it delivers" [Mad+02]. Subsequently, a definition of these core attributes and their quantification possibilities is given.

Reliability can be described as the ability of a system or component to continue performing a designed function [Hor+22] over a specified time. Thus, it can be described by the Mean Time To Failure (MTTF), the average amount of time the item is operating correctly before it fails, for a non-repairable system or component [Mis08]. This property is usually measured over a specified time window, notably the deployment time of the item. For that time window it is assumed that the item is functional at start and will cease to operate eventually, after some amount of time has passed. The probability that the item reaches this fail state is the failure probability. The failure probability can be defined by a function $F(t) = P(T \leq t)$ for a given point in time t and the item's lifespan T. Given the previous determination, the function evaluates to 0 at $t \leq 0$ and trends to infinity (see also [MP10]). The reliability of an item is essentially the reverse of its failure probability and thus it can be measured by subtracting the failure probability from the maximum probability: $R(t) = 1 - F(t)$. Regarding an item with an exponentially distributed occurrence of failures, $F(t)$ and $R(t)$ can be rewritten as:

$$F(t) = 1 - e^{-\lambda \cdot t} \tag{2.1}$$

$$R(t) = 1 - (1 - e^{-\lambda \cdot t}) \tag{2.2}$$

with λ being a constant failure rate (1/MTTF).

If we can distinguish from the cause of the failure state, it can be possible to determine and separate between a safety- and a security-based reliability. In [Mad+02] the authors introduce a *Mean Time To Security Failure (MTTSF)*, describing the average amount of time the (non-repairable) system is operating correctly before it fails due to a security incident. Based on this, independent failure probabilities and reliability values could be derived, analogously to the above definition.

Reliability and safety are related concepts but they are not identical and sometimes even converse. While commonly a system ought to be safe and reliable, it can also be one without the other. Recalling, a safe system is protected against the occurrence of catastrophic events. Considering a brake assist with two different modes for the emergency object detection; fine and coarse. The fine mode is overcautious and thus produces many false positives, however, also captures all actual emergencies. This mode is unreliable due to many false detections but safe because no emergency remains undetected. The coarse mode does not suffer from false positive detections and is reliably detecting most objects, yet not all of them. This mode is highly reliable as it only captures actual emergencies, though, it is not safe as it fails to capture all objects and thus the entry of catastrophic events becomes more probable.

Maintainability or also repairability describes the ability of the system and its components of restoring the operable condition after a failure [Mis08], which is usually measured as a probability over a specified time window (deployment time). Unlike repairability, which only concerns the actual repair-time, maintainability concerns the down-time of the system, including repair-, administrative- and logistic-time. This detail is however not further concerned in the course of this thesis. Consequently, it can be described by the Mean Time To Repair (MTTR) of the system. Maintainability is an essential property to avoid or decrease the total down-time of the system and thus enhancing the system's availability. Analogously to the failure probability, maintainability can be described by a probability under the assumption that the down-time can be described by a random variable T_s [MP10]:

$$M(t) = P(T_s \leq t) \tag{2.3}$$

Accompanying the density function and a maintainability rate can be defined.

Besides internal repair mechanisms, it is important to note that the system can be maintained by the outside, by e.g., performing software updates. This aspect is also highly relevant for security as the quality of cryptographic functions relies on up-to-date implementation of protocols and algorithms and more importantly a

sophisticated key management. In that matter, outdated software, communication protocols, keys and passwords are the main contributions to successful security attacks. As stated in [TD21], the maintainability of a system characterizes its adaptability to the detection, elimination and prevention of failures, thus fault tolerance mechanisms (more in this regard can be found in the upcoming Section 2.4) are key to establishing a maintainable system.

Availability is important for both safety and security. It describes the property of the system for being accessible, operational and providing the intended information when required, relating to the proportion of time the system is available *to use* [TD21]. Thereby, it is dependent on the system's security, reliability and maintainability, by relating the number of occurred failures (Mean Time Between Failures (MTBF), uptime) to the ability of the system to deal with them adequately (MTTR, downtime) over time: $Availability = \frac{MTBF}{MTBF+MTTR}$. Hence, the system is available for a certain amount of time, that may be extended due to maintenance actions, but the time for performing these must be abstracted from the actual uptime of the system. The probability that a system is available at time t is given by the function $V(t)$:

$$V(t) = (\text{System is operational at time } t) \tag{2.4}$$

This formula depends on the system model and can equally be determined by a boolean or stochastic modelling according to [MP10].

In a system without maintenance, the availability equals the reliability, however in repairable systems the availability is always higher than the reliability [MP10]. From the security perspective, it is moreso essential that the access to the system and thereby retrieved information is provided to authorized persons only. This is a key measurement to protect the system against various kinds of malicious attacks.

2.4 Failure Behaviour and Tolerance Strategies

Even though a plethora of approaches and techniques exist to enhance and verify the correctness of a critical system by its design, there is no such thing as zero risk for failures. Besides the fact that hardware faults caused by environmental effects like radiation cannot be prevented, one major reason is that we are still unable to produce error-free software [Tor00]. Especially in consideration of highly complex systems like the autonomous vehicle, it must be assumed that fault avoidance

measures, that have the purpose of handling faults by preventing their occurrence in first place, are not capable of eliminating all faults [Hor+22]. Critical and complex systems that rely on extensive software usage are thus required to be aware of faults and failures instead and provide mechanisms to handle them adequately. Therefore these systems usually implement a so-called fault tolerance strategy. This strategy defines the system behaviour in the presence of faults, in relation to the consequent impact for its functionality and its capability of maintaining a safe state. The system's functionality is defined as its expressed behaviour in interaction with its operating environment [Sto+22], meaning the behaviour that is relevant to perform its intended task. The safe state is defined as a state or operational mode (according to ISO 26262 [18]) where the system does not pose an unreasonable risk [Sto+22]. The most common fault-tolerance strategies to maintain a safe state are fail-safe, fail-operational and fail-degraded. Stolte et al. [Sto+22] define these as follows:

> ***Fail-safe***: *A system is fail-safe in the presence of a fault combination if it ceases its specified functionality and transitions to a well-defined condition to maintain a safe state.*
>
> ***Fail-operational***: *A system is fail-operational in the presence of a fault combination if it can provide its specified functionality with at least nominal performance while maintaining a safe state.*
>
> ***Fail-degraded***: *A system is fail-degraded in the presence of a fault combination if it can provide its specified functionality with below nominal performance while maintaining a safe state.*

In other words, a fail-safe system can tolerate zero faults, however manages to gracefully transition into a safe failure state. A fail-operational system is capable of tolerating one or more faults without any performance loss. A fail-degraded system can tolerate system failures while maintaining a safe state, however is only capable of maintaining a degraded functionality. A system is called fail-unsafe, if it does not deal with faults at all. Thereby a safe-state cannot be ensured.

2.5 Assessment Paradigms

Assessment approaches can be described by a plethora of attributes. Earlier we saw that approaches can be of quantitative or qualitative nature. Coming from another point of view, approaches can be categorized by different paradigms which

are derived from the characteristics of the applied techniques. To provide a common basis, the following gives an outline on the paradigms *analytical*, *numerical*, *symbolic* and *simulation-based*.

Assessment approaches consist of several layers where different techniques and methods are applied. These layers are most simply divided into the modelling layer and the solution layer, though further divisions may be made. On the modelling layer, the concerned system is represented by a mathematical model which can be, among others, an automaton or a mathematical equation, depending on the approach and the aspired solution technique. On the solution layer, a mathematical method is applied to solve or evaluate the system in consideration of some property describing the evaluation goal. Analytical solution procedures are characterized by being exact and solvable by pencil and paper, while numerical solution procedures are approximate, because an exact solution cannot be obtained in finite time and usually computer support is required. Following from that, numerical solution procedures include algorithms such as the numerical integration and the power method/iteration. However, simulation-based methods also characterize as numerical, because they only approximate the problem's solution. Symbolic solution procedures lie somewhere in between, since they solve analytical problems with computer support by simply being capable of performing more steps than a human by hand. As an example, automatic theorem provers are considered in this field. Confusingly, the field of numerical model checking concerns the application of numerical algorithms, while statistical model checking is essentially the application of Monte-Carlo simulation. Yet, this is not contradictory, as simulation can be seen as a special class of numerical methods. Figure 2.6 provides an overview on the given specifications. The paradigm of an approach may be

Figure 2.6: Assessment Paradigms Overview

determined by its applied solution procedure. However, in practice a distinct classification can be very difficult. Purely analytical or numerical approaches are rare due to their poor practicality in capturing complex real-world systems, and thus several methods from different fields are used. Though, the general nature of an

approach can be determined, which is important to describe incidental characteristics. Sargent [Sar94] defines an analytical assessment approach as an approach that models a system as a mathematical equation and uses an analytical equation or a numerical algorithm to solve it. The numerical algorithm can become necessary in cases where the analytical solution cannot be solved in a reasonable amount of time. A simulation-based approach by Sargent [Sar94] and also Zizka [Ziz05] is defined as an approach that takes a conceptual model of a real system and emulates its operating behaviour under consideration of the functional relations. Results are received by analysing the thereby obtained data set further. Thus, also simulation is not necessarily free of analytical methods, as they may become necessary to obtain the aspired results. Simulation is capable of delivering realistic results with a possible simplified application and modelling, but it can be extremely time-consuming and limited in regards of repeatability and additional analysis options [MS01]. Analytical assessment, on the other hand, excels by delivering very precise results [MS01], without requiring multiple calculations (simulation runs). However, a mathematical description of the model is required and the approach may suffer from unrealistic and restrictive assumptions, or complex modelling issues like exponential state growth. To make use of the advantages of either approach, they can be combined into a hybrid model, where the mathematical model combines identifiable simulation and analytical models, as further specified in [Sar94].

2.5.1 Probabilistic Model Checking

Probabilistic model checking is a well-established technique of computer-aided verification for the modelling and analysis of stochastic systems. A prominent method to model stochastic systems are Markov chains or Markov processes, named after the Russian mathematician Andrey Markov, which describe a sequence of possible events (states and transitions). Their key property is that the probability for reaching a state only depends on the present state, which is called *memoryless* and is referred to as the Markov property. This technique makes use of an analytical modelling. In theory it can be solved analytically, however due to its complexity, numerical algorithms or occasionally simulation-based solutions are applied.

Varying types of Markov chains exist. The simplest one is the Discrete-Time Markov Chain (DTMC) which is essentially a Kripke structure[11] where all trans-

[11] A transition system proposed by Saul Kripke based on a graph structure where the nodes represent reachable states and the edges define transitions between them.

itions are equipped with a probability [Kat16]. Formally the DTMC is given by a tuple (S, s_{init}, P, L) (see also [KNP10]) with

- S denoting a set of states,
- $s_{init} \in S$ being the initial state,
- $P : S \times S \rightarrow [0, 1]$ defining the transition probability matrix such that $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$
- and $L : S \rightarrow 2^{AP}$ a labelling function that assigns each state a set of atomic propositions AP.

When modelling a system as a DTMC each state represents a configuration of the system, e.g., the health status of its individual components. A transition marks the probability for changing from one state into another in a step of discrete time. A path in the DTMC is a sequence of possible states. Probabilistic model checking of DTMCs additions the classical analysis (e.g. determining steady-state behaviour) by providing the ability to reason about path-based properties [KNP10]. Therefore, a probability space is defined over the set of all paths through the model. The desired properties are expressed using temporal logic, namely, Probabilistic Computation Tree Logic (PCTL) which is specified in [HJ95]. The PCTL formulas are built making use of atomic propositions, propositional logic connectives and operators that express time and probabilities. Thereby, path properties and state properties can be expressed that hold universally for the entire chain (operator A) or existential for selected paths (operator E). This allows us to formulate and evaluate a plethora of system properties, from qualitative properties such as the reachability of a specific state, to quantitative properties that determine the probability for reaching certain state(s) is below a given threshold. For example, $P^{<0.5}\,[F^{\leq 5} fail_A]$ expresses that the failure probability of some component A is below 50% within 5 steps of discrete time (transitions of the DTMC). While qualitative properties can be reduced to reachability problems that can be solved using standard graph search algorithms for finite-state Markov chains, quantitative properties are determined by linear equation systems [Kat16].

Certain systems profit from a more realistic modelling of time. For example, in reliability engineering the failure and repair behaviour of components is usually described in rates rather than probabilities. To cover these models, one possibility is to extend DTMCs to CTMCs. According to Katoen [Kat16] this can be done by formulating a function $r : S \rightarrow \mathbb{R}_{\leq 0}$ that assigns each state $s \in S$ the rate of a negative exponential distribution that determines the residence time in s. Then the transitions between two states $s, s' \in S$ are described by : $R(s, s') = P(s, s') \cdot r(s)$. The probability to reside maximally t time units in state s is

expressed by $1 - e^{-r(s)\cdot t}$. Thus, the probability for moving from state s to s' in the time interval $[0, t]$ is given by:

$$\frac{R(s, s')}{r(s)} \cdot (1 - e^{-r(s)\cdot t}) \tag{2.5}$$

Formally the CTMC is a tuple (S, s_{init}, R, L)

- S denoting a set of states,
- $s_{init} \in S$ being the initial state,
- $R : S \times S \to \mathbb{R}_{\leq 0}$ defining the transition rate matrix
- and $L : S \to 2^{AP}$ a labelling function that assigns each state a set of atomic propositions AP.

A path in the CTMC is a sequence that alternates between states and time delays. The temporal logic PCTL does not offer support for specifying continuous time properties, however, it can be used to evaluate the properties of the underlying DTMC. For time-based properties, the temporal logic Continuous Stochastic Logic (CSL) can be used which extends the features of PCTL by introducing a time-bounded and a steady-state operator as specified in [Azi+96]. This extends the analysis options by model checking for timed reachability, such as computing a specific behaviour after a precisely defined amount of time has passed, e.g., $P^{=?}\ [F^{\leq 5}\ fail_A]$ which expresses the probability that a component A fails within 5 months. Of course the actual unit of the F operator is depending on the rates set in the model, hence, in one model this time bound is related to months, in the other to hours, minutes or seconds and so on. Let G denote the set of states that we aim to reach in our property ($P^{=?}\ [F^{\leq t}\ G]$). As stated by Katoen [Kat16; Kat13], the properties of timed reachability can be characterized by the Volterra integral equation system. To solve such a property, he states that a function $x_s(\delta)$ associated for every state s and every non-negative real value δ can be defined as follows:

1. if G is not reachable from s, $x_s(\delta) = 0$ for all δ
2. if $s \in G$ then $x_s(\delta) = 1$ for all δ
3. and otherwise:

$$x_s(\delta) = \int_0^{\delta} \sum_{u \in S} R(s, u) \cdot e^{-r(s)\cdot y}\ \cdot\ x_u(\delta - y)\ \ dy \tag{2.6}$$

Case one regards states which do not have the possibility to transition to a state $s \in G$ and case two all states which are itself part of G. Case three is the interesting case where the viewed state s can reach a state in G with a transition, but it does not belong to G itself. Here the integral equation is required, as s may have several successor states and there might be several routes in which s are capable of reaching G and all contribute to the probability of fulfilling the property. Solving this integral equation is, nonetheless, a non-trivial and inefficient task, but it can be reduced to the computation of transient probabilities. These transient probabilities have shown to be solved efficiently by the application of numerical or statistical model checking (see also [Kat13]). In [You+06] a comparison of both techniques for the transient analysis of stochastic systems is performed. As mentioned earlier, statistical model checking makes use of Monte-Carlo simulation and sampling. Thereby problems are solved by statistical hypothesis testing, which essentially consists of producing random paths and checking whether the property that shall be verified holds. Numerical model checking, on the other hand, makes use of numerical algorithms, such as uniformisation, which makes it more exact than statistical model checking, though usually far more memory intensive. Plenty of tools exist that enable modelling and model checking of different types of Markov chains, for example, PRISM[12] and Storm[13].

2.5.2 Stochastic Simulation: The Monte-Carlo Method

The general idea of (computer) simulation is to imitate the behaviour or operation of some real world system. Therefore, the conceptual model of the real-world problem is instantiated by a simulation model, representing the key characteristics of the real-world object or system. Performing the simulation, the evolution of the model is viewed usually over a defined interval of time, or after specific operations have been processed. Based on the observations of the model an average behaviour, allegedly picturing the real-world system, can be assessed to make statements and predictions of the real-world system.

Computer simulation has broad application fields and thus various types exist. Simulation techniques are widely applied across various industrial and scientific fields. Due to its characteristics, simulation can be especially beneficial for use-cases where the real-world system does not exist (yet), e.g., during the development phase, or if the simulation is intended to predict the future. For example, the authors of [SB06] apply simulation to predict climate change scenarios and abbreviate risks for the occurrence of catastrophic events. In this case, historic

[12] https://www.prismmodelchecker.org/ visited on 6th July 2022
[13] https://www.stormchecker.org/ visited on 6th July 2022

real-world data exists which is used to predict outcomes for the future. Furthermore, simulation can be used as a less cost- and time-intensive measure as an addition to the regular real-world testing. Thereby, it can help to meet the existing requirements while keeping the costs on a reasonable level. This work focuses on simulation that solves stochastic systems like Monte-Carlo simulation. These kinds of approaches approximate the solution for the input system or equation. Thereby the quality of the results, hence accordance to the actual solution, depends on the number of performed iterations. This is explained by the law of large numbers[14]

Monte Carlo Simulation is a special type of numerical simulation. It excels by being capable of modelling systems which cannot easily be predicted due to the intervention of random variables and uncertainties, by modelling the probability of different outcomes [Ken21]. Technically this is achieved by repeated random sampling and making use of a statistical evaluation to compute the final prediction of the MCS.

First of all, a deterministic model that closely resembles the real-world system must be developed. Various types of models can be used, to provide some examples, Reliability Block Diagrams (RBDs) in combination with a state-based diagram for the simulation are used in [Hei+19] and in [Car+08] neural networks and are used to describe the structural behaviour of the system. These models dictate the system events that may occur during the simulation. Then, instead of using average values for (uncertain) system parameters, such as the reliability of a component, random samples are drawn from statistical distributions and used as input values (see also [Ray08]). This distribution is typically identified based on the historical data, experience or expert knowledge of a particular input parameter's component. Fitting routines are applied to this set of historical data to identify the most suitable probability distribution. A frequently applied method is the maximum likelihood estimation. Afterwards, repeated random sampling is performed by the generation of random numbers from this distribution that represent specific values for the parameter [Ray08]. The quality of the random number generator is a deciding factor for the accuracy of the prediction. Each simulation run produces output parameters which form one particular outcome scenario. Several simulation runs are performed yielding a data collection of possible results. In the end, statistical analysis is applied to this collection to obtain the final result. This could be, for instance, the computation of the average behaviour of a component during all considered simulation runs.

[14]The average of the results obtained from a large number of trials should be close to the expected value and tends to become closer to the expected value as more trials are performed [Dek+05].

Figure 2.7 visualized this concept.

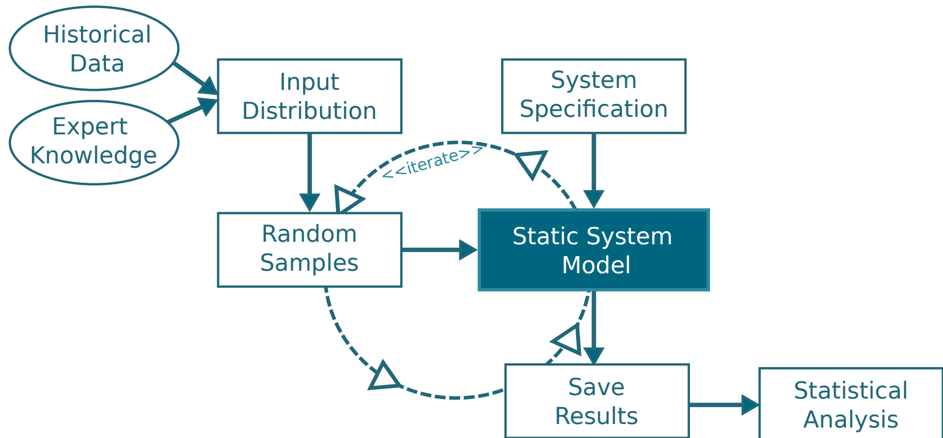


Figure 2.7: Monte-Carlo Simulation Concept

In this way, MCS also allows us to assess systems with high uncertainty and where the input parameters are depending on various external factors. Consequently, it is a particularly well suited method for modelling the impact of risk [Ken21], and thus prominently applied in the finance sector and in reliability engineering. However, it also finds great utilization in other fields such as medical science where it can be used for dose finding in clinical trials (see [OC91]). While it has the typical benefits of a numerical approach in terms of scalability especially in regard of large and complex systems, it can become extremely time consuming, e.g., in cases where very low (failure) probabilities are evaluated [Car+08]. One possibility to overcome this problem would be the application of *weighted* MCS [MP10], where the simulation paths are assigned with additional probabilities as weights. Numerous possibilities on performing MCS exist. Individual solutions based on common and high-level programming languages such as C++ and Python are frequently developed. Further, for many popular languages supporting frameworks and libraries already exist. Additionally to that, specific tools and tool integrations such as Matlab's Simulink[15] or Palisade's @Risk[16] can be used.

[15] https://www.mathworks.com/discovery/monte-carlo-simulation.html visited on 6th July 2022

[16] https://www.palisade.com/risk/default.asp visited on 6th July 2022

3 Modelling

As we saw earlier, modern vehicles are controlled by an orchestration of numerous distributed components, each fulfilling a narrowly defined task while communicating with one another via dedicated bus systems. While all these components fulfil relevant tasks, this work is interested in the components that establish or impact the core functionality. From this point of view, we can distinguish between components that are mandatory to this functionality, e.g., the engine control unit, non-mandatory like the infotainment system and components that lie somewhere in between, such as sensors providing essential data to mandatory components. Which components are required to perform the driving task can depend on the environment the vehicle is placed in. This makes it reasonable to distinguish between these environments and their unique requirements for the vehicle in analysis. For instance, urban driving scenarios require various sensors and intelligent applications to detect traffic lights, diverse street signs, pedestrians etc., and ultimately take profound decisions to manoeuvre around them safely. On the motorway (highway, autobahn) on the other hand, traffic lights and pedestrians are absent together with a lower variation in street signs. This leads to an assumably simpler scenario in regard to perception and decision making requirements. Considering that, the modelling must offer a way of defining mandatory components flexibly.

Earlier it was presented how safety and security condition one another. For instance, the safety-wise failure of a cryptographic module increases the vulnerability of the component and thus may favour security attacks. The other way around, a component that was successfully occupied by an attacker endangers the correct operation of other connected components by, e.g., producing unexpected outputs. With the goal of investigating the failure behaviour of the system with respect to its capability of preserving its core functionality in presence of diverse component failures and incidents, it is essential that an accurate modelling considers these intertwined effects. We can achieve this by firstly modelling component failures and incidents in parallel, and secondly modelling the implications of safety to security and vice versa.

In order to satisfy the strong safety requirements from legislators and the public, autonomous vehicles are usually required to behave *fail-safe* (see Section 2.4) as

the minimum requirement. Meaning, in case of emergency (e.g. the loss of a mandatory component) it must be ensured that the system can always be brought into a safe state. This safe state is characterized as a state or situation where the safety of the passengers and the surroundings, i.e., other traffic participants and the infrastructure, can be ensured. Given that, we can see that the safe state definition actually depends on the current driving scenario. For driving on the motorway manoeuvring to and halting on the verge is a typical safe state. However, in urban driving we can assume that such a spot may not be easily accessible due to crowded streets and pavements, parked cars or traffic structures like intersections that forbid a quickly performed parking manoeuvre. Therefore the requirements for the system to reach the safe state can differ, which must be considered in the modelling and its subsequent analysis.

In the following sections the nature of these briefly introduced aspects is analysed further. The goal is to collect, abstract and determine the relevant safety and security effects native to automated and autonomous vehicles, to provide a transition from the real-world system into a formal modelling. This modelling defines the frame for the later following analysis and is formally defined in the upcoming Chapter 4. Therefore the modelling shall include the varying criticality levels of components (see Section 3.1), the individual failure and incident behaviour of components as well as the resulting implications to their intertwined safety and security conditions (see Section 3.2), and the existing mechanisms to enhance the tolerance against faults (see Section 3.3). In Section 3.4 possible assessment goals and strategies based on this analysis are discussed.

3.1 Component Criticality and Redundancy

The vehicle's components are of varying importance for establishing the core system functionality. With respect to that, we can define their criticality to receive a basis for reasoning about the system's capability of staying operational to at least reach its defined safe state. Taking a look at the previous example; we can see that there is a large amount of comfort components in vehicles (infotainment system, body control module etc.) that are not mandatory for driving safely and thus can be considered uncritical. On the other side, we have components like the engine control unit that are crucial for performing this task and thus are required to operate correctly and reliably. We can categorize these components as critical. Given that, a failure of a non-mandatory component does not directly lead to a penalty of the core functionality, yet a failure of a mandatory component disables this functionality entirely. However, there are also non-mandatory components

that indirectly impact this functionality by perhaps enabling communication interfaces, or providing functionally relevant data to mandatory components. This makes it inadequate to view failures and incidents of mandatory components solitarily and instead we must include the existing dependencies in our criticality consideration. In that sense, the absence of an output due to a defect or a corruption of a component can have an adverse effect on the component that relies on it. For example, if the object detection application does not receive sensor data, it cannot perform its designated task even though it is not flawed itself. In order to meet the high safety and reliability requirements of critical systems, components that contribute to a critical task are often redundantly designed. These can be critical components themselves such as the computing units of the SDS as pictured in Section 2.2, but also sensors which are not mandatory themselves but provide necessary outputs. For an accurate representation we have to take care of these considerations when defining component criticality.

To support a system-wide definition for required components and required outputs, a criticality definition is proposed that collects all components relevant to preserving the core functionality. These components are required to be operational. If a component of the criticality definition is redundant, all of its twins must be included in the definition as well, with at least one of these redundant components being required to operate correctly. Provided that, the criticality definition specifies the minimum functionality which is required to reach the safe state and ensure a fail-safe behaviour. Since the safe state and also the components to preserve this functionality can differ in varying driving modes, the criticality definition must be adjustable. In this way, a mode for urban driving can differ from a mode effective when driving on the motorway, even though the considered system remains unchanged.

3.2 Safety and Security

Rather than inspecting the safety and security of particular components, the goal is to view their interplay; in how security incidents or safety failures of individual components will contribute to a loss of the entire system, i.e., the defect or corruption of a critical component. To achieve that, firstly each component is abstracted to a black-box that pictures its functional behaviour, failure behaviour, functional dependencies with other components and its accessibility including security-wise vulnerability based on the existing connections and messaging behaviour. In general the inability of a component to provide its designated function can be the result of a safety failure or a security incident. Even though the cause and output

behaviour differ, the consequences for other, connected components are identical: The output, if there is still any, cannot be trusted to be correct any more. Therefore it is essential that we model safety and security effects in parallel. Additionally to that, the interaction between safety failures and security incidents must be considered by including both; the implications of security incidents to safety and the implications of safety failures to security in the modelling. Thereby the modelling concerns only the viewed system and we explicitly disregard from any implications due to information leaving it. Hence direct implications to, e.g., other vehicles in the environment such as the effect of provided V2X data are excluded from the modelling. In regard of incoming data from the outside, merely the accessibility effects of components are modelled. Functional effects are neglected by virtue of requiring further knowledge of the sending system such as availability, failure behaviour and so on. Summarizing, privacy concerns and functional implications to and from non-system components lie out of the scope of this methodology.

Concerning security, an attack is viewed as the successful exploit of a component's vulnerability, which results in the attacker capturing the component and obtaining the ability to manipulate its internal program. The result is twofold: On the one hand, the manipulated program may compute wrong output data and also cause a malfunction of the subsequent components that rely on the now manipulated output. On the other hand, the attacker may use the captured component as a basis to attack further neighbouring components (e.g. that are placed on the same communication bus or provide some wireless access). Assuming that initially the attacker only resides in the environment of the vehicle, the attacker may propagate through the system by capturing further components step by step. The probability that a vulnerability is successfully exploited depends on various factors regarding internal implementation details, as well as the strength of the security mechanisms installed and operating in the targeted component. Thus, to determine the vulnerability of the component the hard- and software properties of implemented concepts, used libraries and communication protocols must be investigated in regard of possible vulnerabilities and exploits. As a basis, for example, the CVSS standard can be used which provides a general scheme for rating security vulnerabilities. Furthermore, approaches like [Wan+10; KC13] explore entries in Common Vulnerabilities and Exposures (CVE) databases to calculate attack probabilities. Alternatively, metrics as in [Mun+15] and risk assessment approaches as in [Lon+19] can be used to identify the exploitability of components and topologies, providing a basis for establishing probabilities for security attacks. The main difference to safety is that security incidents always require a human player, the attacker, to actually take advantage of the discovered vulnerabilities. Thus, some approaches, for instance [KWS21], take the motivation of the attacker

into account. This is reasonable, because we can assume that, e.g., the vehicle of a celebrity or politician is more likely to be attacked than the vehicle of a normal civilian. In the modelling each component must be annotated with a rate that describes the feasibility to perform a successful attack in terms of a probability under the presumption that an attacker has some physical or virtual access (WLAN, Bluetooth etc) to it. Independent from the internal protection mechanisms of the component, this probability depends on the active security mechanisms provided by other components, for example, hardware modules such as Hardware Security Modules (HSMs) providing a *root of trust* or software-based cryptographic libraries enabling secured communication. The outage of such components providing these security mechanisms will increase the vulnerability of all components relying on them.

Regarding safety the exact functionality or timing behaviour of a component is abstracted by viewing the component as an abstract computation unit receiving some input, providing some output and executing an internal program. Thereby we can assume that a component operates correctly if it executes the designated internal program and all components providing required inputs are operating correctly as well. Thus, its correctness depends on the input provided by other components and the correctness of the component's internal hard- and software. For the modelling, the component's failure behaviour is described by a rate that indicates the probability of occurrence for a critical failure. While in this work the focus lies on the component's functional safety, it is also conceivable that a component itself remains operational, but its functionality deviates from the expected. Imagine a RADAR sensor's cover being obscured by mud. The sensor remains operational but ceases to provide its intended functionality as the radio frequencies are blocked. These kind of failures, similar to physical attacks, lie in the field of Safety Of The Intended Functionality (SOTIF) [22]. Since they have a different source and are commonly of temporary nature, they are not explicitly concerned here. Nevertheless, if we wanted to model them and are able to describe their occurrence as a rate in accordance with the occurrence for failures, we can also cover them by this modelling.

Estimating the risk of a component is subject to a detailed safety analysis of its hard- and software. Typically the failure probability of a component is determined by the probability of hardware faults, which have a tendency to lead to software failures. Hardware faults are of physical nature and are caused by various effects such as wear, vibration, radiation and variations in temperature. This makes them quantifiable by, e.g., fault tree techniques [DT16] and simulation-based approaches [WCH11] and describable by exponential distributions. For pure software faults this is less straightforward. Software faults are by definition faults of the design

which are of non-probabilistic nature. Though, as argued in [KES09] such a fault may only occur if the given input sequence leads to the execution of the faulty part in the source code (which might not happen at all), adding a certain randomness to the in fact deterministic behaviour. Under this assumption we can model software faults probabilistically and if all possible input sequences are known we can determine the software's behaviour statistically. For instance, [KES09] presents different methods for quantifying the software failure probability in critical systems such as in nuclear power plants.

With that we are in a position of including the interaction between safety and security in several ways: Components may turn into a non-operational state due to a successful attack on a data providing component. Further, a component that fails safety-wise is ineffectual in its default functionality but also as an operating platform of the attacker. The vulnerability of a component to security incidents can be increased by the safety failure of a component that provides security mechanisms to it.

3.3 Fault Tolerance

The high complexity of automated and autonomous systems, especially under the application of AI technologies like neural networks, renders classical design approaches that afford to break down the system specification into single elements, having knowledge of all system states at all time, obsolete. In order to meet the high safety requirements regardless of that and thus increase the availability and the practicability of autonomous systems, the trend is shifting from developing fail-safe systems to *fail-operational* systems. Fail-operational systems are capable of tolerating one or more faults without encountering a degradation in performance or the need to immediately enter the safe state (see Section 2.4 or [Sto+22] for more information).

The key measure to achieve such a fail-operational system is the implementation of hard- and software redundancy in combination with a sophisticated failure management process [Hor+22]. Following from that, a realistic system assessment must include the implications of the implemented failure management and repair behaviour, as it can significantly increase its safety and security Key Performance Indicators (KPIs).

As a primary step, the modelling must allow for the redundant definitions of components, or rather the information they provide, as defined in the previous Section 3.1. Secondly, components that have previously been in a non-operational state

due to safety failures or security incidents can turn operational again under predefined conditions that reflect the requirements and the effort of the component's failure management. For this repair behaviour it is assumed that the system is capable of efficiently identifying the non-functional components and the problem source. In practice failure management processes (e.g. FDIR) and IDS (see also [Al-+19]) are capable of performing this task. Owing to the fact that the success of a repair action is dependent on the problem source which varies highly between safety and security failures, as well as hardware and software problems, Chapter 6 provides an overview on existing methods and derives a generalized scheme to extend the modelling.

3.4 Assessment Options

The described modelling abstracts the essential safety and security effects that exist in critical, complex systems. The goal of this modelling is to build an abstraction layer which allows us to reason about these effects with the aim of predicting statements of the quality and risk of the vehicle throughout its deployment time. Given the emphasis of safety and security dependencies, as well as failure and incident behaviour of the components, qualitative assessment methods can be applied to highlight possible and impossible events. Such an event would be the defect of a component as a result of a failure, the inability of the component to function due to missing inputs or the corruption of a component due to an attack given by the architecture and predominant dependencies. In combination of these events we can investigate questions such as "Can a specific component be successfully occupied by an attacker?", "Is it possible that first component A, then component B gets occupied by an attacker?" or "Is my system still operational if a certain component ceases to operate correctly?". Additionally to that, we can analyse the paths an attacker has to take to propagate through the system in order to reach a specific component. In the big picture, this can support the finding of critical dependencies and attack paths that violate predominant (safety and security) requirements.

However, most of these questions can be answered by observing the modelled system structure alone. Further, qualitative assessment methods deliver binary answers, but safety and security properties are not of binary nature; A system is never 100% safe nor secure, it is safe and secure to a certain extent (under certain conditions). Quantitative assessment methods, on the other hand, allow us to mathematically calculate specific properties and estimate the actual system behaviour. On the basis of failure and attack probabilities of the individual

components, the analysis of undesired events could be extended by an evaluation of their occurrence probability in regard to passed deployment time or happened events. Thereby the deployment time is an important factor – in practice the manufacturer provides guarantee and support for a certain envisaged life span of the vehicle. Thus, from the view point of the manufacturer, who is essentially applying such an assessment, it is important that the vehicle satisfies its requirement of staying safe and secure at least until its specified *end of life* is reached. In contrast to qualitative assessment methods, quantitative assessment methods let us answer questions of the type "What is the probability that first component A, then component B is occupied by an attacker" or more generally "What is the probability that our system fails within the maximum lifespan?". Based on the latter we can then obtain further safety parameters of the system such as its reliability and availability. These results can be used as supporting evidence that under the made assumptions the system stays safe, available and reliable to a calculated probability (extent).

4 Formalization

This chapter presents the mathematical methodology that formalizes the modelling of the previous Chapter 3. The goal is to capture the intertwined effects of safety and security in critical and interdependent systems, and offer a solution procedure that enables the evaluation of risk and quality properties. In the first Section 4.1 the modelling layer is described. This layer specifies the model, a so called dependency graph, to instantiate a real-world system in terms of the previously determined properties. Thereby, the actual system is abstracted to a directed graph, with nodes representing components and links their safety- and security-wise relations. The second Section 4.2 concerns the solution layer. The here presented solution procedure is a type of Markov analysis that makes it possible to evaluate various system properties of either quantitative or qualitative nature. The section begins with a formal transformation from a dependency graph into a corresponding Markov chain (an instance of a CTMC). Afterwards the evaluation of the Markov chain to obtain the aspired system assessment is discussed and the scalability of the approach is analysed.

4.1 Dependency Graph

The dependency graph is a directed graph that is meant to represent a real-world system abstracted to its functionality and existing safety and security dependencies of individual components. The method was firstly introduced in [RH20b]. The general idea is that system components are mapped to nodes and the various interrelationships between them are abstracted to links. The profundity of the abstraction is specific to the use case and the desired assessment goal. In our vehicle context, a node can range from being a single ECU or sensor, over to a more complex unit such as an entire infotainment system. But the abstraction is not bound to physical components. It can desist from the actual hardware and, for instance, define a software library or a software application, to give an example, a firewall as a node. Consequently, the node breaks down the considered system part in a functional way: *The functionality is active/provided as long as no defect/attack on the underlying hardware occurred.*

Besides the nodes that result from the system abstraction, a special node Env is introduced reflecting the environment of the system. In this environment other traffic participants such as connected vehicles, the manufacturer or maintainer (server or personal in workshop), but also possible attackers reside. The Env-node is essential to model any connections inheriting from these players connected to internal interfaces provided by the connectivity components (see also Section 2.2.4). This design builds the basic structure to model security incidents by reflecting possible attack interfaces.

Formally, the graph is given by a set of nodes $\mathcal{N} \cup \{\mathsf{Env}\}$ and a set of links $\mathcal{L}$. In order to distinguish between the semantics of the different node connections, three link types are separated. This is done by annotating each link $l \in \mathcal{L}$ with a type $\tau(l) \in \{\mathsf{Fct}, \mathsf{Reach}, \mathsf{Sec}\}$ expressing the characteristics of the dependency:

- Fct-links model the functional safety dependencies of the components, for instance, an actuator relying on the input of one or more sensors. These links are essential to express that a component ceases to operate correctly, if the functionally required output produced by another component is not available anymore (due to a functional defect or corruption by an attacker).
- Reach-links express the connectivity between individual components, indicating that the source component is able to directly communicate and command the target component. A Reach-link originating in the environment node expresses that the target component is able to be accessed from the outside directly, such as a connectivity component that is providing a Bluetooth or WLAN interface. In this way, Reach-links describe the entry points of an attacker and the routes he may take to propagate through the system: Starting with nodes directly connected to the Env, the attacker may target further nodes by abusing the existing Reach-links to neighbouring nodes.
- Sec-links build a selective class of functional dependencies that captures the protective side of security. They are used to express that a component is providing some sort of security guarantees to another component. For example, an HSM provides cryptographic mechanisms to other connected components. The idea is that the chances for a successful attack are lower as long as these security guarantees can be provided.

Figure 4.1 shows a very abstracted scheme of an autonomous vehicle, covering the different node and link types of a dependency graph. Node n_5 represents the telematics unit of the system that is accessible from the outside suggested by the Reach-link coming from the Env-node. In reality, the telematics unit may receive software and parameter updates which are forwarded to the computing

unit n_1, expressed by the Reach-link between them. This incoming connection is secured by a firewall, which is modelled as n_4 and is providing security guarantees to the telematics unit n_5. Node n_1 represents the centrepiece of the system, the computing node, that uses the sensor's n_2 input to determine the operations that shall be performed by the actuators n_3. Thus, a Fct-link from the sensor to the computing unit is drawn to suggest that the computing unit is relying on this data. Another Fct-link from the computing unit to the actuators suggests that the actuators rely on the processed sensor data. Further, to express that the computing unit is commanding the actuators, a Reach-link between them is drawn.

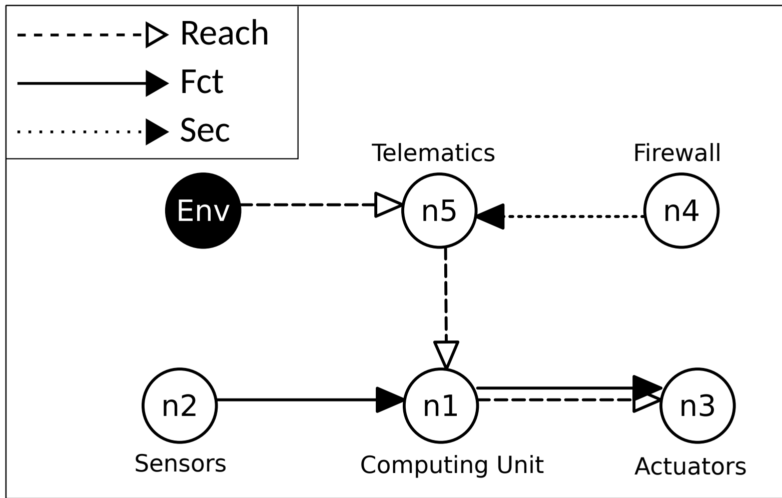


Figure 4.1: Motivational Example of a Dependency Graph

While between the regular system nodes all three link types can be arranged depending on the actual system dependencies, Env-nodes are restricted to outgoing Reach-links. The reason for that is simple: Targeting or originating Fct- or Sec-links would suggest that the system is providing/receiving some functionality to or from the outside. This could be a possible relation, for instance, a GPS sensor relying on satellite information. But if we were to model this behaviour, we would require information on the satellite's functional behaviour, in order to model that this information could be flawed or not provided due to a connection loss. Given that, we would actually have to model the satellite as a system node. This shows that originating or targeting Fct- and Sec-links cannot express a valid semantic in our model. A similar situation arises by Reach-links targeting the Env-node. This modelling would express that information is leaving the system. We could use this to express privacy concerns of the component. However, this is, as discussed (see also Section 3.2) outside the scope of this modelling. To describe the failure

behaviour and attack vulnerability in a quantitative way, each node has occurrence rates for safety failures and security attacks.

Definition 1 gives a complete specification of the dependency graph.

Definition 1 *A* dependency graph $\mathcal{G} = \langle \mathcal{N} \cup \{\mathsf{Env}\}, \mathcal{L} \rangle$ *is a finite, directed graph where*

- $\mathcal{N}$ *is a list of nodes representing the individual components of the architecture.*
  - *each node* $n \in \mathcal{N}$ *is annotated by a failure rate* $r_n^{Safe} \in \mathbb{R}_{\geq 0}$ *and*
  - *each node* $n \in \mathcal{N}$ *is also annotated by a function* $Sec_n : 2^{Src^{\mathsf{Sec}}(n)} \to \mathbb{R}_{\geq 0}$ *mapping sets of components providing security mechanisms to a rate of a successful attack on* n*, where* 2^X *denotes the powerset of* X*.*
- $\mathcal{L} \subseteq (\mathcal{N} \cup \{\mathsf{Env}\}) \times \mathcal{N}$ *is a set of links between nodes. Each link* $l \in \mathcal{L}$ *is annotated by a type* $\tau(l) \in \mathcal{T}$ *with* $\mathcal{T} = \{\mathsf{Fct}, \mathsf{Sec}, \mathsf{Reach}\}$ *being the set of link types.*

Let $t \in \mathcal{T}$*,* $\mathcal{L}^t(n) = \{\langle n', n \rangle \mid \langle n', n \rangle \in \mathcal{L} \wedge \tau(\langle n', n \rangle) = t\}$ *denotes all links of* $\mathcal{L}$ *to* n *of type* t*.* $Src^t(n) = \{n' \mid \langle n', n \rangle \in \mathcal{L}^t(n)\}$ *denotes all nodes that are the origin to these links.*

It is important to note that while for safety each node can simply be annotated by a rate r_n^{Safe}, indicating its probability to experience a critical failure, for security a more complex definition is required. Therefore the Function Sec_n is defined, marrying the rate that expresses the probability for an attack on the concerned node with the rate of security guarantees provided by connected nodes via Sec-links, because the provided security guarantees lower the probability for an attack. Firstly all links that target the viewed node are collected by $\mathcal{L}^t(n)$. Then they are filtered by their type with the function Src^t and t being Sec. Ultimately the collected security guarantees are subtracted from the initial attack rate.

For example, Figure 4.2 shows a connectivity component connected to the Env-node with a Reach-dependency to some ECU. This ECU is protected by two nodes, n_2 and n_3, whereas n_3 provides encryption functionalities and n_2 authentication. Since n_4 only has one incoming Reach-link $Src^{\mathsf{Reach}}(n_4) = \{\mathsf{Env}\}$, $Src^{\mathsf{Sec}}(n_4) = \emptyset$ and $Src^{\mathsf{Fct}}(n_4) = \emptyset$. n_1, though, is reachable from n_4 so that $Src^{\mathsf{Reach}}(n_1) = \{n_4\}$. Furthermore, it receives security guarantees from n_2 and n_3, hence $Src^{\mathsf{Sec}}(n_1) = \{n_2, n_3\}$. Consequently, Sec_{n_4} merely depends on its own attack probability, while Sec_{n_1} additionally depends on the guarantees provided by n_2 and n_3.

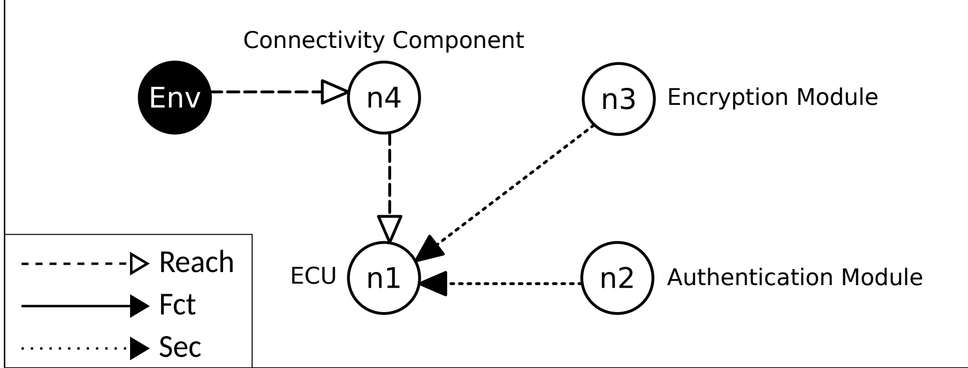


Figure 4.2: Security Function Example

Since nodes can be in normal operation, or in experience of a safety failure or a security attack, they actually can adopt different states. Suppose it is possible to identify and distinguish between failures and attacks, three states of a node can formally be defined as $\forall n \in \mathcal{N}.\ s(n) \in \{\mathsf{ok}, \mathsf{def}, \mathsf{corr}\}$ with the following semantic:

- ok expresses that the component represented by the node is operating correctly, hence it operates as specified. This is expected to be the initial state of every node.
- def indicates that the represented component does not work correctly anymore due to an internal hardware or software failure and, as a consequence, the output (if still generated) cannot be relied on.
- corr means that an attacker has successfully occupied the represented component and is capable of controlling it. As a consequence the attacker misuses its communication properties and it can no longer be trusted to be producing the correct and expected output.

This allows us to reason about the impacts of undesired events for the system and for connected nodes. The collection of these node states comprises the state of the dependency graph which is reflecting the health of the modelled system. This state is essentially a configuration of all its node's states, formally given by the subsequent Definition 2.

Definition 2 *Let $\mathcal{G} = \langle \mathcal{N} \cup \{\mathsf{Env}\}, \mathcal{L} \rangle$ be a dependency graph. $\Sigma = \{\mathsf{ok}, \mathsf{def}, \mathsf{corr}\}$ is the set of different* node states. *A* state *of the dependency graph $\mathcal{G}$ is a mapping $s : \mathcal{N} \to \Sigma$. $S_{\mathcal{G}}$ denotes the set of all states of $\mathcal{G}$ (if $\mathcal{G}$ is known from the context we simply write S).*

Let $s \in S_{\mathcal{G}}$*,* $\sigma \in \Sigma$*, and* $n \in \mathcal{N}$ *then* $s[n \leftarrow \sigma]$ *is the state defined by* $s[n \leftarrow \sigma](n) = \sigma$ *and* $\forall n' \in N \setminus \{n\}.\ s[n \leftarrow \sigma](n') = s(n')$*.*

In a dependency graph with four nodes as in the previous example of Figure 4.2, the state is defined by the state of its components, i.e. $(s(n_1), s(n_2), s(n_3), s(n_4))$. Recalling the definition of the attack rate of a node Sec_n, it becomes clear that this occurrence rate actually depends on the current state of the dependency graph. In the example, the ECU n_1 is most vulnerable if the encryption module and the authentication module (n_2 and n_3) have failed, e.g. $(\mathsf{ok}, \mathsf{def}, \mathsf{def}, \mathsf{ok})$, because this would prevent the provision of any security guarantees. Thereby, it is generally irrelevant whether these nodes are in the state def or corr, however, in the pictured example they can never turn corr due to the given dependencies. Further, n_2 and n_3 can provide different amounts or quality of protection. This means that the vulnerability of n_1 can be different in states $\langle \mathsf{ok}, \mathsf{def}, \mathsf{ok}, \mathsf{ok} \rangle$ and $\langle \mathsf{ok}, \mathsf{ok}, \mathsf{def}, \mathsf{ok} \rangle$. For instance, with the ECU potentially being the target of malicious commands, the validation of the commands' authenticity is most relevant and can be seen as a higher security guarantee than its encryption.

Since in reality components can be of varying criticality, as discussed in the previous chapter, some nodes may cease to operate without having an effect on the performance of the system, while others directly lead to a system failure. We instantiate the discussed criticality definition by defining a so called *mode of operation.* A mode of operation is represented by a boolean formula connecting nodes that are guaranteeing the correct operation of the system in their current state by making use of logical and-/or-operators. It evaluates as true if the system is operational, indicated by the health states of the included nodes, false otherwise. The included nodes are essentially all nodes that are mandatory to provide the system's safe state. In reverse conclusion, the mode of operation can be used to derive failure states, which are all states where the mode is not satisfied. Definition 3 specifies the mode of operation formally.

Definition 3 *Let* $\mathcal{G} = \langle \mathcal{N} \cup \{\mathsf{Env}\}, \mathcal{L} \rangle$ *be a dependency graph. A mode of operation is a propositional formula* ϕ *over* $\{\wedge, \vee\}$ *and logical variables* $\{\hat{n} \mid n \in \mathcal{N}\}$ *(representing the individual nodes of* $\mathcal{N}$*).*

A state $s \in S_{\mathcal{G}}$ *induces an assignment* μ_s *on* $\hat{\mathcal{N}}$ *by:* $\mu_s(\hat{n}_i) = true$ *iff* $s(n_i) = \mathsf{ok}$*. A state* $s \in S_{\mathcal{G}}$ satisfies ϕ *iff* $\mu_s(\phi) = true$*.*

Figure 4.3 shows another very basic scheme which could be part of an automated vehicle to explain the definition of the mode of operation. In the centre we see a perception unit n_2 that receives data from the outside via V2X n_1 that can

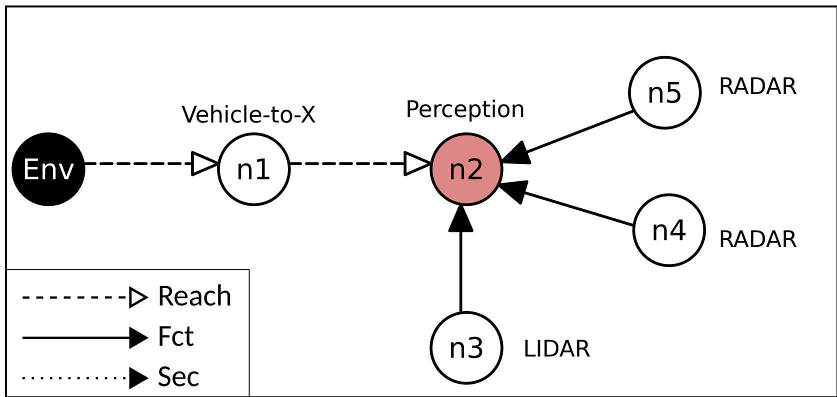


Figure 4.3: Nodes with Different Criticality

possibly change the system behaviour is, however, not functionally required. This could be perhaps relevant data transmitted from other vehicles like their current driving speed. Functionally important data is provided to the perception unit by three sensors n_5, n_4 and n_3. In this graph, the set of nodes is given by $\mathcal{N} = \{n_1, n_2, n_3, n_4, n_5\}$. A possible and convenient mode of operation could be defined by $\phi = \hat{n}_2 \wedge (\hat{n}_5 \vee \hat{n}_4) \wedge \hat{n}_3$. This declares that the perception unit n_2 must be operational and is relying on one of the RADAR sensors (n_4 or n_5) and the LIDAR sensor n_3 to be operational. Thus the RADARs are designed to provide redundant information. The node of the perception unit is marked red here to highlight that it is critical for the system itself. While nodes n_3, n_4 and n_5 constitute to this required system functionality, they are not mandatory themselves. This is a different semantic from the system's point of view, but formally the nodes are handled identically by being part of the mode of operation. Thus the red highlighting is only syntactical sugar, emphasizing the criticality of a component. To support the system's change of criticality in different (driving) situations, multiple independent modes of operation for the same system can be established.

4.1.1 Application Example

Until now we only observed very basic examples of dependency graphs to visualize the fundamental concepts. For real systems, the dependency graph can quickly become very complex. Figure 4.4 shows an exemplary illustration of the dependency graph of an abstracted but realistic architecture of an autonomous vehicle. The implemented architecture is based on the system design of a modern (autonomous) vehicle in accordance to the one described in the earlier Section 2.2.

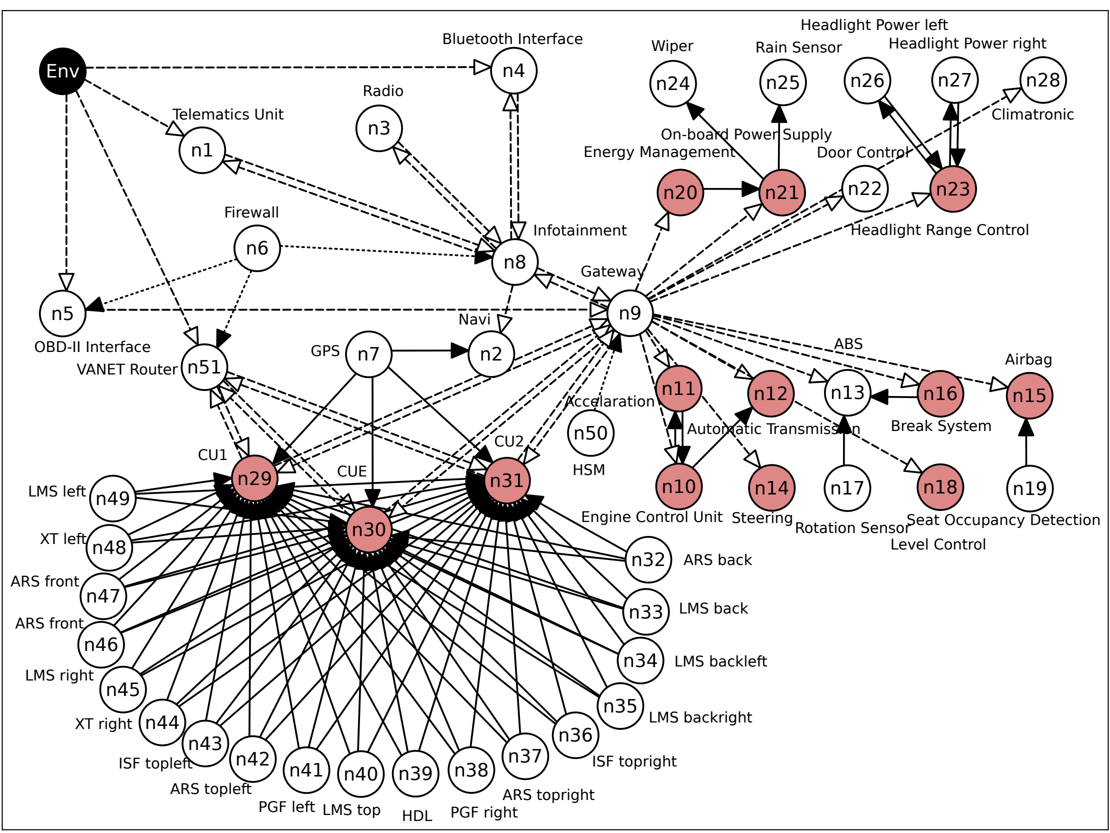


Figure 4.4: Dependency Graph of an Autonomous Vehicle

On the right hand side the legacy components (Section 2.2.2) of the comfort (upper) and motor (lower) CAN bus are outlined. Here several nodes were highlighted red to emphasize their criticality. For example, the vehicle cannot function without an operational engine control unit n_{10}. Furthermore, functional dependencies exist, for instance, the ABS n_{13} is relying on the information of the rotation sensor in the wheels n_{17} to fulfil its designated task. Since in reality these components lie on a bus system which is connected to a gateway (see also Section 2.2.1) for inter-bus communication, all of these represented components were modelled reachable from the gateway node n_9. This is necessary to express that the represented legacy components can receive commands from other components via the gateway, which may be exploited by an attacker. Contrary to the architecture presented in Section 2.2, the gateway of the SDS and the legacy components is modelled as a single gateway for simplicity. To create an exact model, Reach-dependencies between the legacy component's nodes would need to be drawn to reflect the commanding-semantic among the components on the bus. However, since Reach-dependencies

are used to express vulnerability to security attacks based on existing commanding interfaces and all of the nodes are already reachable through the same path, these dependencies can be omitted without consequence. This would be different if one of the legacy component's node was connected through another path from the Env-node, providing for a separate attack path which would increase the probability for attacks.

Because of the gateway's central role in communication (and reachability), an HSM n_{50} was modelled to secure it by the supply of cryptographic mechanisms. The semantic here is as follows: As long as the HSM is in a state ok, it is actively providing security guarantees to the gateway and thus lowering the probability for successful attacks on it. This indirectly lowers the attack probability for subsequent nodes, hence the ones representing legacy components, as they can only be attacked once the gateway node has been occupied by the attacker.

On the lower left, the SDS and its required sensors are modelled. The SDS itself is represented by three critical system nodes n_{29}, n_{30} and n_{31} representing $CU1$, CUE and $CU2$, just like the system architecture described in Section 2.2.3.

To accord the redundancy definition of the proposed architecture, where at least one of the computing nodes has to be operational, we can simply write $\hat{n}_{29} \vee \hat{n}_{30} \vee \hat{n}_{31}$ in the mode of operation. The reliance on sensor data is modelled by various Fct-links from each sensor node to each CU node. Here redundancy definitions exist, because not all sensors are required to be operational to perform the basic driving task. For example, we may only require one of the high dynamic-range cameras (PGFs) to be operational which we express by adding $\hat{n}_{41} \vee \hat{n}_{38}$ to the mode of operation. The consequences of a degraded functionality, hence the degradation in quality and extent if, e.g., only the emergency CU is still working or redundant sensors have failed, cannot be covered by this model. Further, details such as the application switching between the CUs in failure cases (between active-hot and active instances as described in the in Section 2.2.3) cannot be captured on this level either. However, it is possible to define separate modes of operation for the system assessment. For instance, in some driving modes like highway driving in comparison to urban driving, only a smaller, selected group of sensors may be required and thus the system can stay operational through multiple sensor failures, while it would not in a different driving mode. Furthermore, one could assume that in highly critical scenarios we actually want to have at least one regular computing node ($CU1$ or $CU2$) and the CUE as a fallback system active.

On the upper left of Figure 4.4 the various connectivity components (see Section

2.2.4) are modelled. The telematics unit n_1, the Bluetooth interface n_4, the OBD-II interface n_5 and the VANET router n_{51} provide interfaces for the communication with the outside and are thus directly targeted by Reach-links from the Env-node. These are the nodes an attacker can attack in the first step, before corrupting any system nodes. Based on their assumed messaging behaviour outgoing Reach-links to other nodes have been set. In this way, the propagation of information but also potential malware is modelled. For instance, Reach-links from the telematics unit to the infotainment and from the infotainment to the navigational system are established. This path exists, because it is conceivable that the telematics unit receives updated map information that is further deployed in the navigational system via the infotainment. Additionally, Reach-dependencies between the infotainment n_8 and the gateway n_9 were modelled, which are necessary to express the distribution of parameter or software updates of the legacy components by the manufacturer via the telematics unit. While the infotainment is already reachable via the described paths, it is further reachable through the gateway node. Contrary to the case between the legacy component's nodes, this Reach-dependency cannot be neglected, because it expresses the existence of another Reach-path. We can see that the OBD-II interface n_5 is also reachable, however, does not target the infotainment but the gateway. This path potentially employs a different attack probability and thus cannot be omitted. With the same idea of the HSM, a firewall is installed as node n_6 that is providing security guarantees to the infotainment, OBD-II interface and the VANET router nodes.

4.2 Dependency Markov Chain

With the formalization of the dependency graph Chapter 3's essential safety and security effects of real-world systems can be captured effectively. However, to analyse these effects and make the dependency graph assessable we need to *put it into operation*, emulating the real-world system's behaviour. Therefore the occurrence of failures and attacks, indicated by the defined occurrence rates for every node, has to be viewed in regard to the elapsed time of operation. The role of the dependency graph is to act as a framework by dictating which effects are possible and defining the consequences for the system's functionality. The analysis goal is to observe how an initially fully operational system degrades in health over time due to failures and incidents and eventually reaches a critical state where the functionality cannot be preserved any more. Technically this means that our interest lies in evaluating quantitative properties that mark the reaching of undesired states. In the dependency graph, these states are substantially all states that do not satisfy the mode of operation. To achieve that, we are required to transform the

dependency graph into a quantitative model. The Markov chain (see also Section 2.5.1) makes an excellent candidate for several reasons: Due to its state-based modelling, we can easily represent the states of the dependency graph by mapping them to the states of the Markov chain. Furthermore, the occurrence of safety failures and security incidents can be expressed by the transitions marking state changes of the Markov chain. Additionally to that, an accurate modelling of time can be induced if we make use of CTMCs. As a key feature, Markov chains support the automatic evaluation by probabilistic model checking which can quickly become mandatory by the growing complexity of input models and also for sheer comfort.

The following sections concern the transformation of a dependency graph into a corresponding CTMC, subsequently referred to as a *Dependency Markov Chain*. Therefore Section 4.2.1 formalizes the transformation, Section 4.2.2 discusses the formal consequences for the scalability and Section 4.2.3 describes the evaluation idea and process.

4.2.1 Transformation

The CTMC is instantiated as a Dependency Markov Chain $\mathcal{DMC}$ in regard of a dependency graph and its predominant mode of operation. Looking back at the definition of Section 2.5.1, a CTMC is a probabilistic automaton that essentially consists of a set of states (with a defined initial state) and transitions between those, where occurrence is defined by the transition rate matrix. For the transformation, each health state of the dependency graph ($s \in S_{\mathcal{G}}$) is simply adopted by a state in the Markov chain. Because our interest lies in finding the first point in time where the system ceases to operate, we can disregard from consecutive failure states and model them final, so they have no outgoing transitions. This lets us comprise these failure states to a single Fail state that marks the state where *things went sideways*. For simplicity a transition in the $\mathcal{DMC}$ only represents the state change of a single node. This helps keeping the number of possible transitions on a reasonable level. Considering the rather low probabilities for failures and incidents of components, this interleaving of potentially concurrent events seems to be a realistic assumption. Thereby, transitions based on safety failures (r_n^{Safe}) leading to the node's def state and transitions based on security attacks r_n^{Sec} resulting in the corr state of the represented node are separated. Figure 4.5 illustrates the $\mathcal{DMC}$ concept. From the initial state on the left, a safety failure transition (top) may be taken on behalf of rate r_n^{Safe} turning the fictive node n_1 def, or a security attack transition (bottom) defined by rate r_n^{Sec} may be taken turning it corr. Depending on the underlying graph, various transitions may occur that eventually lead to

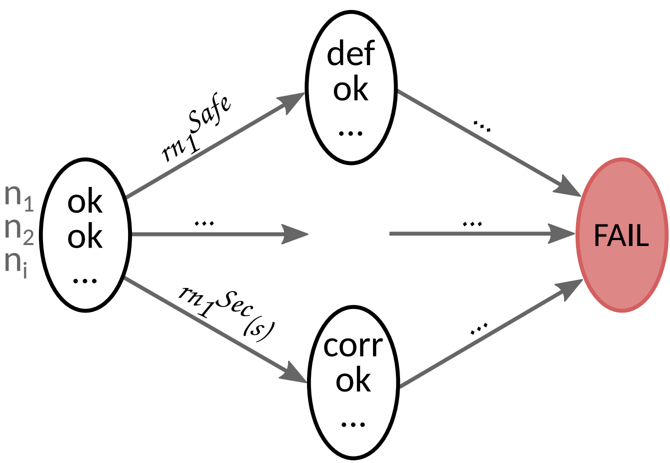


Figure 4.5: Conceptual Illustration of the $\mathcal{DMC}$

undesired states summarized by Fail. Definition 4 gives a complete and formal specification on the transformation of a dependency graph into its corresponding CTMC under a given mode of operation.

Definition 4 *Let $\mathcal{G} = \langle \mathcal{N} \cup \{\mathsf{Env}\}, \mathcal{L}\rangle$ be a dependency graph with states $S_\mathcal{G}$ and nodes $\mathcal{N} = \{n_1, \ldots, n_k\}$. A CTMC $\mathcal{C} = (\mathbf{S}, \mathbf{s}_{init}, R, L)$ is a* Dependency Markov Chain *($\mathcal{DMC}$) of $\mathcal{G}$ w.r.t. a mode of operation ϕ. Let $\bar{S} = \{s \in S_\mathcal{G} \mid \mu_s(\phi) = true\}$.*

- $\mathbf{S} = \bar{S} \cup \{\mathsf{Fail}\}$
- $\mathbf{s}_{init} = s \in S_\mathcal{G} \cap \mathbf{S}$ *with* $\forall n \in \mathcal{N}.\ s(n) = \mathsf{ok}$
- $L(\mathsf{Fail}) = \langle Fail \rangle$ *and* $L(s) = \langle s(n_1), \ldots s(n_k)\rangle$
- $R : S_\mathcal{G} \times S_\mathcal{G}$ *is given by (for* $n \in \mathcal{N}$*)*
  - $R(s, s[n \leftarrow \mathsf{def}]) = r_n^{Safe}$ *if* $\mu_{s[n\leftarrow \mathsf{def}]}(\phi) = true \wedge s(n) \neq \mathsf{def}$
  - $R(s, s[n \leftarrow \mathsf{corr}]) = r_n^{Sec}(s)$ *if* $\mu_{s[n\leftarrow \mathsf{corr}]}(\phi) = true \wedge s(n) = \mathsf{ok}$ $\wedge\, \exists \langle n', n\rangle \in \mathcal{L}^{Reach}.\ (n' = \mathsf{Env}\ \vee\ s(n') = \mathsf{corr})$
  - $R(s, \mathsf{Fail}) = \sum_{i=0}^{|\mathcal{N}_{Safe}|} r_{n_i}^{Safe} + \sum_{j=0}^{|\mathcal{N}_{Sec}|} r_{n_j}^{Sec}(s)$
    with $\mathcal{N}_{Safe} = \{n \in \mathcal{N} \mid s[n \leftarrow \mathsf{def}] \notin \bar{S} \wedge s(n) \neq \mathsf{def}\}$
    and $\mathcal{N}_{Sec} = \{n \in \mathcal{N} \mid s[n \leftarrow \mathsf{corr}] \notin \bar{S} \wedge \exists \langle n', n\rangle \in \mathcal{L}^{Reach}.$
    $(n' = \mathsf{Env}\ \vee\ s(n') = \mathsf{corr})\}$
  - *and* $R(s, s') = 0$ *for all other pairs* s, s'

To define the Markov states (first item of Def. 4) in accordance with the operational states of the graph as mentioned, firstly the set $\bar{S}$ is built. It consists of all dependency graph states where the mode of operation evaluates as true. This set of states and an explicitly labelled Fail state are adopted in the Markov chain. The initial state, marked as $\mathbf{s}_{init}$, is defined to be the state where all nodes are ok, following the idea that at start the system is at full health. The third item defines the labelling function so that the initial state can be written as $\langle \mathsf{ok}, ..., \mathsf{ok} \rangle$ and all failure states as $\langle \mathsf{Fail} \rangle$. The transition rate matrix R of the $\mathcal{DMC}$ assigns rates to each Markovian state pair, specifying the transitions as we saw before, by reflecting the health state change of a single node. To handle the varying transitions we are required to distinguish different cases: In the first two cases we view transitions that do not lead into the Fail state, hence, where the succeeding state still satisfies the mode of operation. Due to divergent conditions, defects and corruptions are handled separately. The third case represents all missing transitions which are leading into the Fail state.

Precisely, the first subitem defines the transition into a Markov state where a concerned node turns def. This transition is allowed regardless of whether the previous state was ok or corr. The probability of the transition is simply adopted by the fixed rate r_n^{Safe} of the node describing its probability for a hard- or software failure (see Definition 1). The second subitem defines the transition into the Markov state that reflects the corruption of the node. This transition has additional conditions: It must only be performed if the current state reflects a state where the node is ok, because we assume that a node that is already irresponsive cannot be successfully occupied by an attacker. Furthermore, the Reach-dependencies dictated by the graph must be taken into account, making this transition only possible in case of a targeting Reach-link from the Env or an already corrupted node ($s(n')$) to the concerned one exists. The probability for this transition is given by $r_n^{Sec}(s)$, which, in accordance to the definition in the graph, is not a fixed rate but a function instantiated with the current Markov state. Formally this function is given in the upcoming Definition 5.

Definition 5 *Let s be a state of a dependency graph $\mathcal{G} = \langle \mathcal{N} \cup \{\mathsf{Env}\}, \mathcal{L} \rangle$ then the attack rate of n in s is defined as $r_n^{Sec}(s) = Sec_n(\{n' \in Src^{\mathsf{Sec}}(n) \mid s(n') = \mathsf{ok}\})$*

Recalling, the corruption of a node is indicated by a rate that describes the exploitation of a vulnerability in dependence to the security mechanisms provided by other nodes with the latter varying if security providing nodes are corrupted or defective themselves. To respect that different rates for an attack exist depending on the system state, the rate must be instantiated by the current state to describe the successful attack of the node in regard of the health of the nodes that are

providing security guarantees. Definition 5 achieves that by making use of the *Src* function as explained in the previous Section.

Now that we defined the safety and security transitions into operational states, we must define all transitions that lead to failure states collected by Fail. Here the computation of the occurrence probability is less straightforward: From the Markov chain perspective all of these previously distinct failure states are summarized into a single state and thereby several transitions to the same state arise, leading to the existence of alternate paths. We can observe this phenomenon in the upcoming Figure 4.8, where all failure states of Figure 4.7 are summarized to single Fail state. According to the definition of the CTMC (see also Section 2.5.1)) alternate paths are required to be summed. To collect the rates of all of these transitions we build two sets $\mathcal{N}_{Safe}$ and $\mathcal{N}_{Sec}$. These sets are used to sum up all safety relevant transitions and all security relevant transitions, just in accordance to the definition of the CTMC design and property evaluation where individual steps in the chain are multiplied, but alternate paths are summed (see Section 2.5.1). Any other transitions that do not accord the specified design are neglected by setting their occurrence rate to zero. Figure 4.6 shows a *minimal*

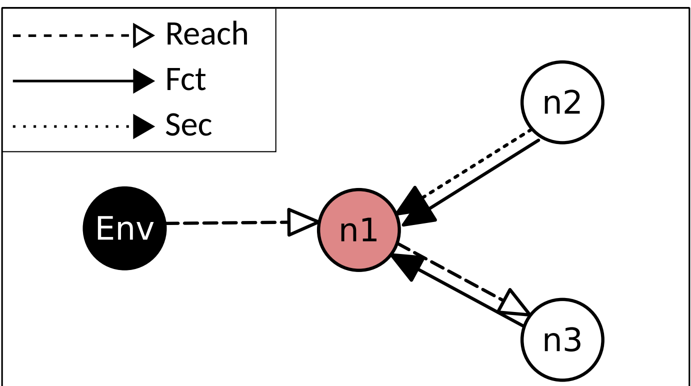


Figure 4.6: Toy Example of a Dependency Graph

working example of a dependency graph that is capable of capturing most of its features. This example is used to show the transformation into a corresponding $\mathcal{DMC}$. The mode of operation is exemplarily set by $\phi = \hat{n}_1 \wedge (\hat{n}_2 \vee \hat{n}_3)$, declaring that node n_1 and at least one of the nodes n_2, n_3 must be operational to provide the desired mode. By specifying that either node n_2 or n_3 must be operational, they are defined to deliver redundant information.

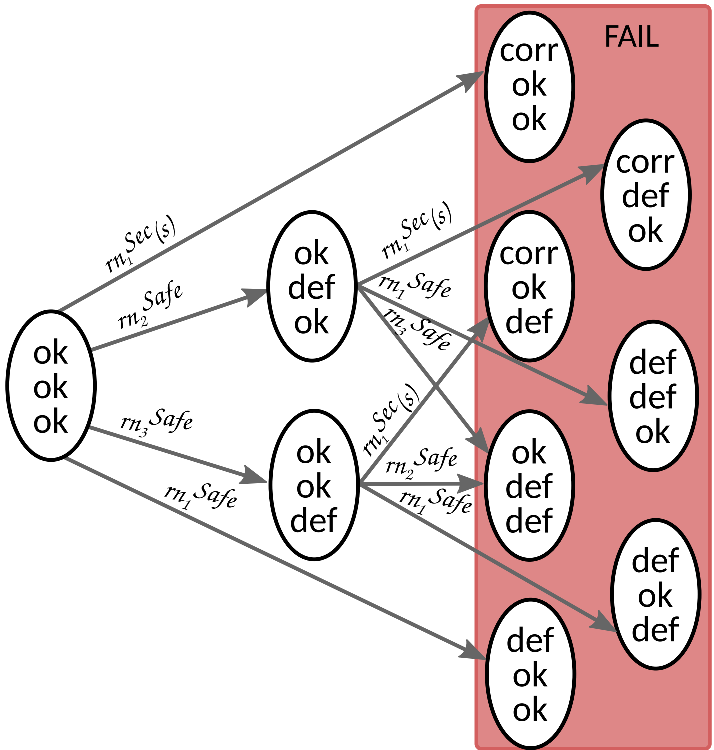


Figure 4.7: $\mathcal{DMC}$ of the Toy Example: Multiple Failure States

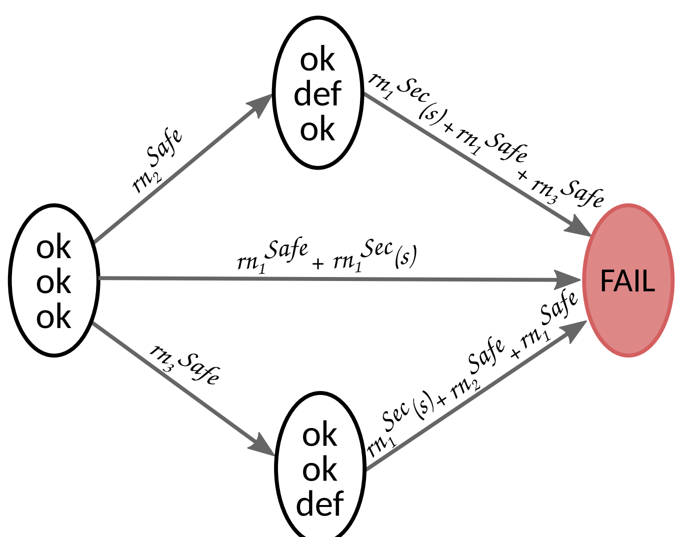


Figure 4.8: $\mathcal{DMC}$ of the Toy Example: Comprised Failure States

Figure 4.7 visualizes the $\mathcal{DMC}$ of the toy example with all failure states and Figure 4.8 the respective summary of these to a singular Fail state. We can see that in accordance with the mode of operation, all states where node n_1, or both nodes n_2 and n_3 are not operational any more, are failure states. Note that due to the graph properties (absence of Reach-link) node n_2 can never be attacked and turn corrupted. Figure 4.7 shows that eight transitions to failure states from three different states exist. By summarizing these failure states, the transitions are summed with respect to their origin state yielding three transitions. For example state $\langle \mathsf{ok}, \mathsf{ok}, \mathsf{def} \rangle$ where the failure or the security attack on n_1 ($\langle \mathsf{def}, \mathsf{ok}, \mathsf{def} \rangle$ and $\langle \mathsf{corr}, \mathsf{ok}, \mathsf{def} \rangle$), but also the failure of n_2 $\langle \mathsf{ok}, \mathsf{def}, \mathsf{def} \rangle$ may lead to failure states.

4.2.2 Scalability

The state space of the $\mathcal{DMC}$ grows exponentially in relation to the nodes of the dependency graph, since every state in the $\mathcal{DMC}$ comprises a health state configuration of all nodes. Considering that we have three health states, this means that the Markov states grow by $3^{|\mathcal{N}|}$ in the worst case. Though, in practice this is not the case as certain properties of the overlaying dependency graph lower the state growth. Yet it always remains exponential. As we could see in the example transformation nodes that are not reachable (are not connected by a Reach-path that originates in an Env-node) cannot be attacked and thus never turn corr. This lowers both the amount of yielding states and transitions. While the goal of the developed method is to evaluate these security effects, in reality not all system components possess commanding interfaces. For components that solely act as data providers this is a reasonable assumption. For instance, the automotive sensors in the application example of the previous Section 4.1.1 are not modelled as reachable. Consequently the state space must be calculated with respect to the amount of non-reachable nodes, as given by the following formula:

$$3^{|\mathcal{N}-\mathcal{M}|} \cdot 2^{|\mathcal{M}|} \text{ with } \mathcal{M} = \{n \in \mathcal{N} \mid Src(n)^{\mathsf{Reach}} = \emptyset\} \tag{4.1}$$

All non-reachable states are defined by the set $\mathcal{M}$. To compute all states of the nodes that can adopt either of the three health states, we subtract the non-reachable nodes from set $\mathcal{N}$ and for these non-reachable nodes we only consider two health states.

In addition to that, the predominant mode of operation in combination with the failure state summary provides for further state space reduction. Recalling, the mode of operation defines whether a state is a failure state and all failure states are comprised to a final Fail state. In this way, dead states (and transitions), that mark

further breakage of an already non-operational system, are omitted and reduced to one single state. Since only nodes that are essential for the system in a sense that their defect or corruption results in failure states, they produce less Markov states than *uncritical* nodes (not providing essential data and not themselves part of the mode). These uncritical nodes are, for example, comfort or not mandatory connectivity components. It follows that we get the worst case of $3^{|\mathcal{N}|}$ states, whenever all nodes of the dependency graph are reachable and uncritical (providing an empty mode of operation). In reality whatsoever, such a system does not make much sense, as semantically it would mean that all modelled components are either uncritical or provide the same redundant functionality.

Given these contemplations, a tiny dependency graph with 3 nodes already produces $3^3 = 27$ Markov states in the worst case, however, as seen in the toy example's $\mathcal{DMC}$ (Figure 4.8) the state space can actually be reduced to 4. In this example, node n_2 is not reachable which reduces the states to $3^2 \cdot 2^1 = 12$. Yet the pictured $\mathcal{DMC}$ without the failure state summary (Figure 4.7) only consists of 10 states. This discrepancy is easily explained: Since failure states are final, the state $\langle \mathsf{def}, \mathsf{def}, \mathsf{def} \rangle$ is neglected, because it may only result from states $\langle \mathsf{ok}, \mathsf{def}, \mathsf{def} \rangle$, $\langle \mathsf{def}, \mathsf{def}, \mathsf{ok} \rangle$ and $\langle \mathsf{def}, \mathsf{ok}, \mathsf{def} \rangle$ which are already failure states themselves. For the same reason the state $\langle \mathsf{corr}, \mathsf{ok}, \mathsf{corr} \rangle$ is neglected. This case is especially interesting, because it reflects the path properties of Reach-dependencies. That is, any state $\langle \ldots, \ldots, \mathsf{corr} \rangle$ can only be entered if n_1 turns corrupted which is yet already a final failure state. The last reduction step is simply the summary to the Fail state where we form an individual state of the 7 failure states as depicted in the example, yielding our 4 state system.

4.2.3 Evaluating the $\mathcal{DMC}$

Now that we are able to build dependency graph-tailored CTMCs we can make use of Markov analysis to answer the desired questions regarding the quality and risk of our modelled system. These are, for example, “What is the probability that our system is still operational after operating for a defined interval of time?” or “What is the probability that our system stays at perfect health after a defined period of continuous operation?”. To show how these questions can be answered, subsequently the application idea of Markov analysis is shown on behalf of the previously introduced toy example equipped with dummy failure and attack rates.

Figure 4.9 shows the transformation of the toy example's dependency graph into its corresponding DMC. For a better readability in the upcoming calculations the states were renamed as depicted and very unrealistic input rates have been chosen.

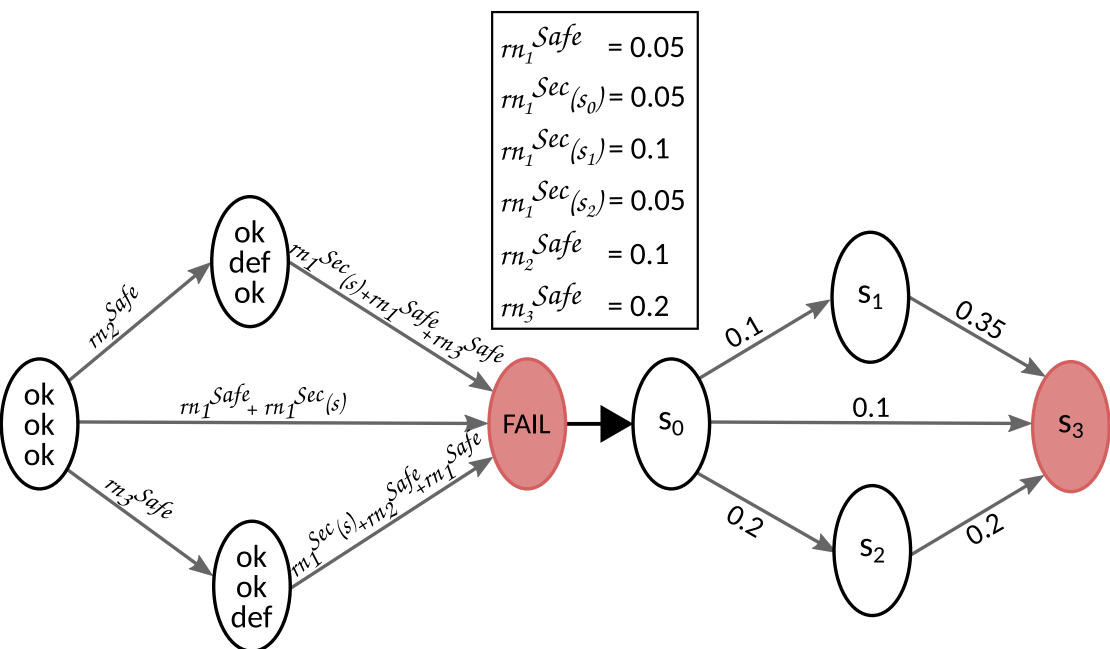


Figure 4.9: $\mathcal{DMC}$ Evaluation Example

Thereby the security guarantees provided by node n_2 were assumed to have a rate of 0.05 and the internal attack rate of n_1 to be 0.1. As a consequence, the rate indicating that node n_1 turns corrupted is different depending on the current state just like defined. In s_0 and s_2 this rate is lower, because node n_2 is still operational and providing security guarantees ($0.1 - 0.05 = 0.05$). In s_1 this rate consists solely of the internal attack rate of n_1.

Lets say we are interested in the probability that our system is still at perfect health after 12 months of deployment and being in continuous operation. To express that property $P_1{}^{=?}$ $[F^{=12}\ s_0]$ is formulated. This is actually a rather simple property, as we only have to view one state (s_0) and calculate the probability that this state is not left. Consequently, no tedious calculations including other states and paths to them have to be considered. Following our previous contemplations (see Section 2.5.1 or [Kat13]) we must calculate $e^{-r(s)\cdot t}$ yielding the probability to be in s after t time units have passed. Since $r(s)$ marks the residence time, but we have the rate matrix $R(s, s')$ given, we are firstly required to determine $r(s)$ for s_0. Due to the relationship of $R(s, s') = r(s) \cdot P(s, s')$, we can calculate $r(s)$ and $P(s, s')$ from the set rate matrix. $r(s)$ is simply calculated by summing all transition rates of s_0: $R(s_0, s_1) + R(s_0, s_2) + R(s_0, s_3) = 0.4$ (see [Kat13] for more information). Then, for example, $P(s_0, s_1) = r(s_0) \div R(s_0, s_1)$ hence $0.4 \div 0.1 = 0.25$.

Calculating the property for $t = 12$ yields $e^{-0.4 \cdot 12} = 0.00082298$. This means that the probability for staying at perfect health after 12 months lies around $\approx 0.08\%$. For visualization, Figure 4.10 shows how this property behaves over time regarding months 1 to 12 in steps of 1 month.

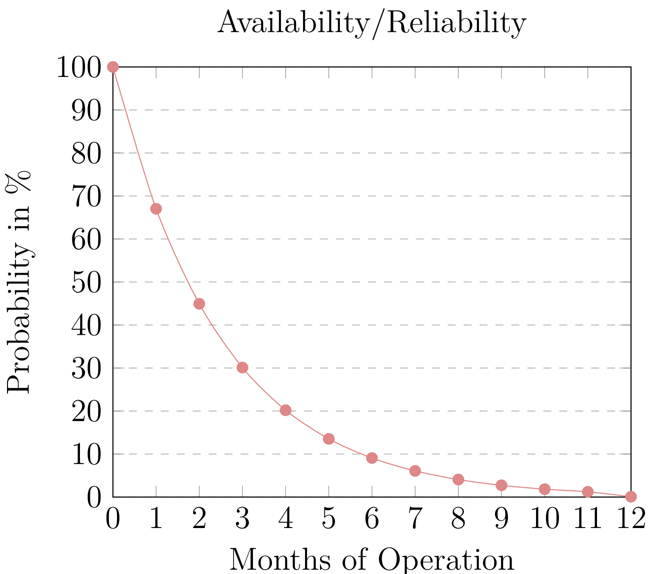


Figure 4.10: Plot: Property P_1 over 12 Months of Continuous Operation

However, viewing only the perfect health is not sufficient in reality. States s_1 and s_2 may indicate a degraded functionality with comfort features not being available any more, yet the system is still functional. Thus, we are more interested in calculating the probability of staying in any of these states, which is substantially the opposite probability of reaching the Fail state. This property is defined by $P_2{=?}\ [F^{=12}\ \mathsf{Fail}]$ ("What is the probability that our system is non-functional after 12 months of continuous operation?"). For this property we do not only have to analyse the probability for staying in a state, but entering it when we firstly are in the initial state (s_0). To enter the Fail state (s_3), several paths consisting of pairs of residence time and transition probability can be taken. Therewith this property falls in the area of timed reachability, which is not easily calculated [Kat13].

To evaluate this property we need to identify the reachability of our state Fail in consideration of some elapsed time. While the reachability alone can be analysed on the basis of the underlying DTMC, for timed reachability we must view the probability to move to the next consecutive state at time y and the probability to

fulfil the property. Therefore the integral equation as discussed earlier (see Section 2.5.1 and [Kat13; Kat16]) must be solved.

$$\begin{aligned} x_{s_0}(12) \;=\; & \int_0^{12} \underbrace{0.1}_{R(s_0,s_1)} \cdot e^{-0.4\cdot y} \;\cdot\; x_{s_1}(12-y)\;\; dy \\ & + \int_0^{12} \underbrace{0.2}_{R(s_0,s_2)} \cdot e^{-0.4\cdot y} \;\cdot\; x_{s_2}(12-y)\;\; dy \\ & + \int_0^{12} \underbrace{0.1}_{R(s_0,s_3)} \cdot e^{-0.4\cdot y} \;\cdot\; x_{s_3}(12-y)\;\; dy \end{aligned} \tag{4.2}$$

$$x_{s_1}(12) = \int_0^{12} \underbrace{0.35}_{R(s_1,s_3)} \cdot e^{-0.35\cdot y} \;\cdot\; x_{s_1}(12-y)\;\; dy \tag{4.3}$$

$$x_{s_2}(12) = \int_0^{12} \underbrace{0.2}_{R(s_2,s_3)} \cdot e^{-0.2\cdot y} \;\cdot\; x_{s_3}(12-y)\;\; dy \tag{4.4}$$

$$x_{s_3}(12) = 1 \text{ (since } x_{s_3} \in P_2) \tag{4.5}$$

Equation 4.2 shows the integral equation of state s_0. The integral is built from 0 to 12 according to the property. Then the rate and the residence time is inserted in the formula accordingly. Because state s_0 can reach all other three states, all the existing paths have to be included by summing them. For s_3 the formula trivially evaluates to 1, as it is the Fail state itself and the property is reached. This example shows that already the analysis of small toy examples can lead to complex solving of integral equations, which is a tedious and inefficient task, making clear that computer support is needed to analyse realistic systems. Probabilistic model checkers offer a great perspective for solving a variety of Markov properties. If we make use of such a tool and are indeed capable of solving the established integral equations, we obtain the failure probability of the system which can be used to calculate further safety parameters.

As an example, the given system is solved using the probabilistic model checker PRISM [KNP11]. The property P_2 yields ≈ 0.8958, meaning by a probability of 89.58% the system is no longer operational after 12 months. As presented in 2.3.2 the reliability and the availability can be directly calculated from this. Hence the reliability of the system is $1 - 0.8958$, which is here the same as the availability, because no system repairs and maintenance were considered. Meaning to a probability of around $\approx 10\%$ our system is still available (operational) after 12 months of operation without any maintenance.

5 Modularization

In the previous chapter the formal model, consisting of dependency graphs and their corresponding CTMC (the Dependency Markov Chain $\mathcal{DMC}$), was presented. The chapter was closed with a description of the evaluation idea and a view on the scalability of the method. Thereby, we could see that an unavoidable exponential state growth in regard of the modelled components (nodes) exists. This is a known downside of Markov analysis or even analytical approaches in general. In regard of the vast amount of components we require to model for accurately capturing an entire automated or autonomous vehicle, this exponential state growth may easily become the bottleneck of our method. As an illustration, the dependency graph of the autonomous vehicle described in Section 4.1.1, Figure 4.4, consists of 51 nodes with 24 of them being reachable. Following the previous scalability consideration (see Section 4.2.2) the corresponding Markov chain yields the tremendous amount of $3^{24} \cdot 2^{27} = 3.790705071 \times 10^{19}$ Markov states (without Fail-state reduction as it is dependent on the predominant mode of operation). Modelling and solving such a Markov chain is a very tedious task that can bring even probabilistic model checkers to their limits due to the enormously required computational resources, in particular memory.

In order to effectively solve systems of this complexity, we are required to find a possibility for decreasing the state space of the Markov chain without greatly curtailing its accuracy. Several optimization options are conceivable. On behalf of the Markov chain, compositional verification and Markov chain abstraction [KNP10] are well-established concepts. The idea of compositional verification is to break the verification of the system into smaller subtasks, often implemented by making use of *Assumption-Guarantee Reasoning* in non-probabilistic contexts, and reduce the verification effort by solving these subtasks independently. Also advances regarding compositional probabilistic model checking have been made [KNP10]. The authors of [HKK13], for example, present an Assumption-Guarantee framework for Interactive Markov Chains (IMC), an extension of classical CTMC by a new transition type. They show how compositional verification in consideration with time-bounded reachability properties can be established. Markov chain abstraction, on the other hand, covers the aggregation of Markov states that are annotated with an interval of rates or probabilities rather than having a single

rate or probability for each state. Thereby, varying techniques to achieve such an abstraction for the diverse types of Markov chains exist. For CTMCs in [Kat+07] a three-valued abstraction method including a model-checking algorithm is presented. [Smi10] applies and implements abstracted CTMCs compositionally in the underlying Markov model of the Performance Evaluation Process Algebra (PEPA) by abstracting each component individually. This is highly beneficial for large Markov models where the state space may be too large to be constructed in first place.

However, our Markov chain is generated in relation to the overlying dependency graph which contains the semantic definition of the system model. Since this information is translated to the Markov chain, it is difficult to extract. Thus, to preserve the integrity of the model it is more reasonable to perform optimizations on behalf of the dependency graph. As a matter of fact, the optimization of graph structures is a long studied subject of theoretical computer science and consequently various concepts and techniques already exist. For instance, graph clustering and graph partitioning concepts can be used to reduce the graph's size by a node-wise division into several smaller pieces. Additionally, graph partitioning is a widely researched field with many different application areas, prominently parallel computing/processing, image processing and VLSI design [BS13] [Bul+16]. The literature separates two main partitioning problems arising from the different application areas: constrained partitioning, where the parts of the partition must be of similar size (e.g. very important for parallel computing/processing, see also [AR06]) and unconstrained partitioning where the parts can be of highly different sizes.

Another option could be to change the solution procedure. For instance, instead of using Markov analysis, simulation-based methods could be used which do not have the trouble of running into state space explosions. While a simulation-based evaluation would bring great benefits regarding the scalability, it also has the disadvantage of generally requiring more modelling effort and being extremely time consuming to reach a similar level of accuracy as the Markov analysis (see also Section 2.5.2).

Inspired by these classical graph partitioning concepts a modularization scheme on dependency graphs is proposed. Thereby, the graph is partitioned into a hierarchical system consisting of one main graph and one or more module graphs. Unlike for parallel computing, in our use case the yielding partitions are not required to be of equal size. Yet, to fulfil its purpose it is necessary that the partition contains enough nodes to decrease the resulting state space in the desired way. In complex systems, consisting of various different components, we can usually observe some sort of functional clustering. Meaning, we can identify component groups that

contribute to a specific functionality that is separate from the rest of the system. For instance, taking a look back at the vehicle example of the previous chapter in Figure 4.4, we can see that the two modelled CAN bus systems (the comfort and the motor CAN) actually make two somewhat distinct and encapsulated subsystems that are solely connected via the gateway. The idea of the modularization scheme is now to identify these rather encapsulated subsystems and separate them as individual dependency graphs. The original system then gets abstracted by modelling these module graphs as nodes. For the evaluation process, the module graph yields the necessary inputs for the node in the abstracted graph and thus requires to be evaluated first. Regarding the solution, two possibilities have been developed: The purely analytical evaluation where both the main and the module graph(s) are evaluated with Markov analysis, and the hybrid where a combination of Markov analysis (the main graph) and MCS (on the module graph(s)) is performed. The decision of allowing the simulation only on certain module graphs is simply to keep the result accuracy high, while respecting subsystem structures that favour a numerical evaluation.

Subsequently the formal basis for the graph division is presented, which was initially published in [RH22]. Afterwards, both solution procedures are described in detail. Thereby remarks on the decision of the evaluation method for the module are made. Finally, heuristics for well-formed modularizations as well as indicators for when a modularization should not be performed (with respect to the result precision) are presented.

5.1 Dependency Graph Division

When executing the modularization, the original dependency graph which is modelling some real-world system, must be split into an abstracted graph and one or more module graphs. In order to make the modularization a sufficient way of staying scalable, it is essential that the graph division is as accurate as possible by reflecting the original dependency graph in the best way imaginable. Therefore, rules must be determined for building the abstracted and the module graph(s) from the original graph.

Given an original system modelled as a dependency graph $\mathcal{G}$, a division is performed that yields an abstracted graph $\mathcal{G}_a$ and a module graph $\mathcal{G}_m$. A comprehensive visual example can be found in Figure 5.1, with an original example system pictured on the left-hand side and its modularization on the right. In the first step we identify a set of system nodes $\mathcal{N}_{sub}$ and their corresponding links that are supposed to build the module graph. In the provided example these are the nodes n_1,

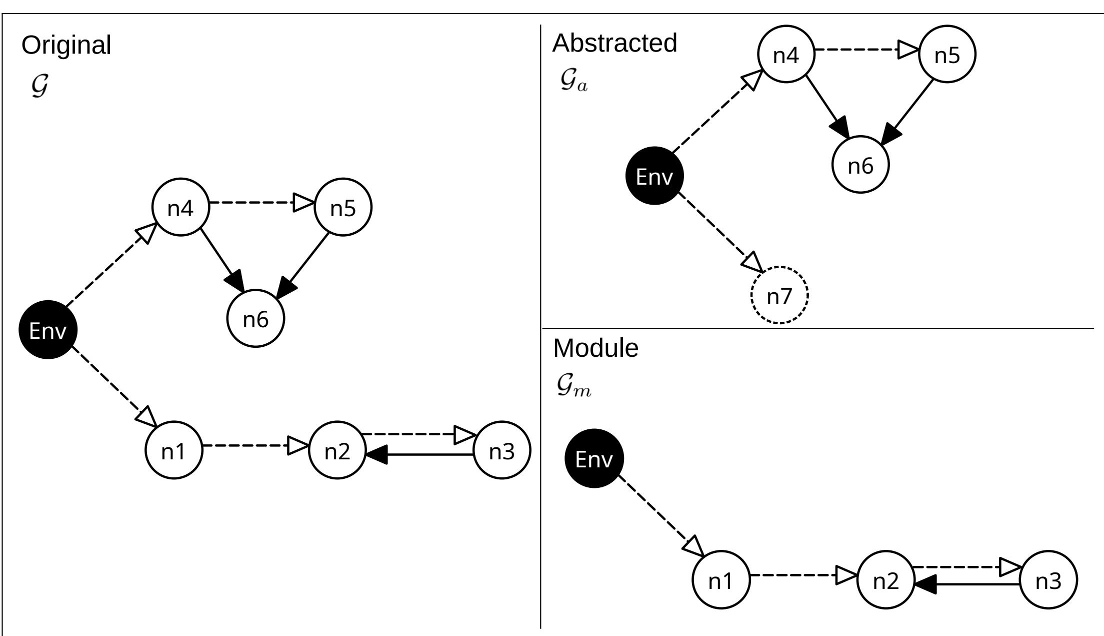


Figure 5.1: Modularization Example

n_2 and n_3. All links between the identified nodes are simply taken over to the module untouched. Since this set of system nodes does not include an Env-node, but the module graph may contain Reach-dependencies, just as it is the case in the pictured example, a new Env-node must be added. This yields the module graph $\mathcal{G}_m$ with nodes $\mathcal{N}_m$ and links $\mathcal{L}_m$. In the next step, we construct the abstracted graph $\mathcal{G}_a$, on the basis of the original graph $\mathcal{G}$, by the removal of all nodes included in the previously identified set $\mathcal{N}_{sub}$ and the links between them. The idea is that because the abstracted graph becomes the main system, which can be evaluated with a decreased amount of modelled components, the removed nodes and links of the module must be considered in a more performant way than in the original model. For that reason, a new node n_{sub} is inserted into $\mathcal{G}_a$, respectively $\mathcal{N}_a$, representing the module graph. In the example this node is referred to as n_7 and highlighted with a dashed line to suggest that it is internally representing a module graph. To accord the original system in the best possible way all links between the module nodes $\mathcal{N}_m$ and the nodes of the abstracted graph $\mathcal{N}_a$ are transferred to n_{sub}. For the example system this is simple: the existing Reach-link from the Env-node to n_1 is transferred to the new module node n_7. This kind of division yields a hierarchical system where the independent evaluation of the module must be performed before the evaluation of the abstracted graph.

Definition 6 specifies the described modularization steps in a formal course.

Definition 6 *Let $\mathcal{G} = \langle \mathcal{N} \cup \{\mathsf{Env}\}, \mathcal{L} \rangle$ be a dependency graph. A set of nodes $\mathcal{N}_{sub} \subseteq \mathcal{N}$ defines a modularization $(\mathcal{G}_a, \mathcal{G}_m)$ by: The abstracted graph $\mathcal{G}_a = \langle \mathcal{N}_a, \mathcal{L}_a \rangle$ is given by*

$$\begin{aligned} \mathcal{N}_a = \;& \mathcal{N} \setminus \mathcal{N}_{sub} \cup \{n_{sub}\} \\ \mathcal{L}_a = \;& \{(n, n') \in \mathcal{L} \mid n, n' \notin \mathcal{N}_{sub}\} \\ & \cup \{(n, n_{sub}) \mid \exists n'.\, (n, n') \in \mathcal{L} \\ & \qquad \wedge n' \in \mathcal{N}_{sub}, n \notin \mathcal{N}_{sub}\} \\ & \cup \{(n_{sub}, n) \mid \exists n.\, (n', n) \in \mathcal{L} \\ & \qquad \wedge n' \in \mathcal{N}_{sub}, n \notin \mathcal{N}_{sub}\} \end{aligned}$$

The module graph $\mathcal{G}_m = \langle \mathcal{N}_m, \mathcal{L}_m \rangle$ is given by

$$\begin{aligned} \mathcal{N}_m \;& = \mathcal{N}_{sub} \cup \{\mathsf{Env}\} \\ \mathcal{L}_m \;& = \{(n, n') \in \mathcal{L} \mid n, n' \in \mathcal{N}_{sub}\} \\ & \cup \{(\mathsf{Env}, n') \mid \exists n.\, (n, n') \in \mathcal{L} \wedge n \notin \mathcal{N}_{sub} \\ & \qquad \wedge n' \in \mathcal{N}_{sub} \wedge \tau((n, n')) = \mathsf{Reach}\} \end{aligned}$$

In reality modularization candidates are often not entirely encapsulated systems. This is only natural considering that complex systems are a composition of multiple subsystems that, however, are required to work together to establish their designated function. These dependencies to neighbouring nodes are taken care of during the replacement of the module graphs by transferring them to n_{sub}. In this way information loss is prevented, though, the dependencies are usually over-approximated. This means that in certain systems, a dependency that originally affected a single node suddenly applies to the entire subgraph represented by the module. A special case is formed by Reach-dependencies to module nodes that express that the target node can be accessed by the origin node. Regarding the vulnerability analysis, Reach-paths that do not originate in an Env-node are senseless, as an intrusion could never take place. Thus we can assume that every existing Reach-dependency is part of a path that originates in an Env-node. To model the access of potential attackers in the module graph, the set of module nodes are extended by an Env-node as discussed, and Reach-links that formerly originated in nodes of the abstracted graph are replaced by Reach-links from that Env-node.

Mode of Operation Accompanying to the modularization of the graph, a division of the mode of operation must be performed. Recalling, the mode of operation

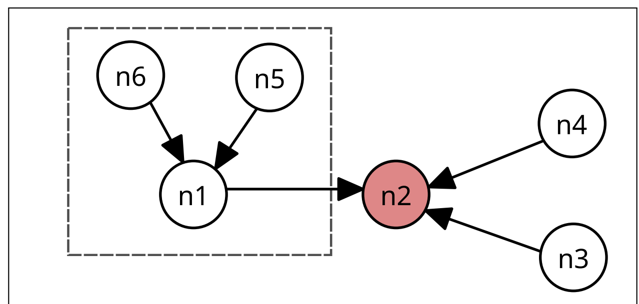


Figure 5.2: Modularization with Functional Dependencies

ϕ defines both, the operational states of the system and the semantics of its functional dependencies, including potential redundancy specifications. On behalf of a mode of operation ϕ for the original graph $\mathcal{G}$, appropriate modes of operation ϕ_a for the abstracted graph $\mathcal{G}_a$ and ϕ_m for the cut out module graph $\mathcal{G}_m$ must be found such that

$$\phi \equiv \phi_a[n_{sub} \leftarrow \phi_m] \tag{5.1}$$

where $\Psi[n \leftarrow \omega]$ denotes the formula obtained from Ψ by replacing each occurrence of n in Ψ by ω.

This is illustrated on behalf of a small example system. Figure 5.2 shows an original system $\mathcal{G}$ and its designated module $\mathcal{G}_m$ in the dashed box. $\mathcal{G}_m$ consists of the node n_1 and the data providing nodes n_5 and n_6. In the original example a functional dependency between n_1 and n_6 exists which must be transferred to a newly introduced module node n_{sub} in the abstracted graph. A possible mode of operation for the original graph could be given by

$$\phi = \hat{n}_2 \wedge (\hat{n}_3 \vee \hat{n}_4) \wedge \hat{n}_1 \wedge (\hat{n}_5 \vee \hat{n}_6)$$

expressing that n_2 (in red), requires n_3 or n_4 as well as n_1 to be operational. n_1 itself requires n_6 or n_5 to be operational. When performing the modularization as defined, ϕ needs to be split to satisfy the equivalence 5.1. For this example appropriate ϕ_m and ϕ_a are given by:

$$\phi_m = \hat{n}_1 \wedge (\hat{n}_5 \vee \hat{n}_6)$$
$$\phi_a = \hat{n}_2 \wedge (\hat{n}_3 \vee \hat{n}_4) \wedge \hat{n}_{sub}$$

whereby the mentioned dependency between n_1 to n_2 in the original graph is represented by a dependency between n_{sub} to n_2 in the abstracted graph. Although this example showed how easily a mode of operation ϕ can be split into appropriate modes ϕ_a and ϕ_m, this is not always the case. In graph structures where nodes of the intended module are not well encapsulated, dividing the mode of operation

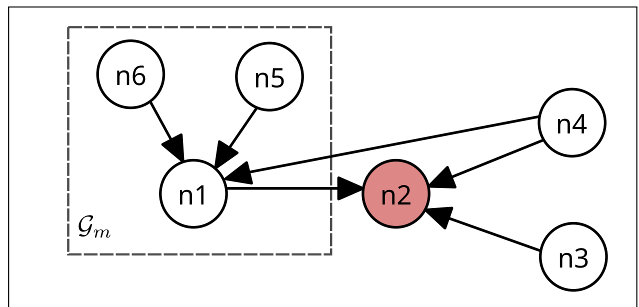


Figure 5.3: Non-modularizable Functional Dependencies

becomes impossible. Reasons for that are functional dependencies or redundancy specifications that cross the border of the module.

To demonstrate this, the previous example is extended by a functional dependency from n_4 to n_1 as pictured in Figure 5.3. This dependency is defined to be redundant to the existing dependencies from n_5 and n_6. Hence, the three nodes deliver redundant data to n_1. With this extension, the mode of operation of the original system adjusts to

$$\phi' = \hat{n}_2 \wedge (\hat{n}_3 \vee \hat{n}_4) \wedge (\hat{n}_1 \wedge (\hat{n}_4 \vee \hat{n}_5 \vee \hat{n}_6)).$$

Performing the split of ϕ' creates a paradox: On the one hand, n_4 is part of ϕ'_a by being a supporting node of n_2 as before. On the other hand, it must also occur in ϕ'_m as a supporting node of n_1. This would require n_4 to be part of the module graph and the abstracted graph at the same time. Summarizing this means, in case a node is occurring multiple times in a mode of operation ϕ, the system can only be transferred into a module if that node occurs always in the context of the later ϕ_m formula (or in an equivalent reformulation of it). This emphasizes that the mode of operation must be determined individually in regard of the original graph and its original ϕ.

5.2 Analysis

The goal of the analysis, to evaluate a quantitative system property with the main focus being the probability that the system reaches a critical failure state, remains unchanged. Due to the hierarchical scheme of a modularized system, any module graphs must be evaluated first, in order to generate the input (failure and attack models) for the module node in the abstracted graph. These failure and attack models express the failure and attack behaviour of the module's components. This

means that the original analysis goal and design has to be rethought for the module. In contrast to our philosophy; the explicit integration of safety failures and security attacks, with the goal of obtaining an overall probability or rate for a system failure, it is now necessary to obtain independent failure and attack rates in the module's evaluation process.

In order to provide additional possibilities for enhancing the scalability two different evaluation ideas were developed. First, the recursive analytical approach, second, the hybrid. In either way, the abstracted graph is evaluated by Markov analyses in accordance with the previous developments. The recursive analytical approach also evaluates the module by Markov analysis. The concrete interplay between the Markov analysis of the module and abstracted graph as well as the internals regarding the evaluation of independent failure models is described in the upcoming Section 5.2.2. In the hybrid approach, the module graph is evaluated numerically with a simulation-based method as presented in detail in Section 5.2.3. The objective of having two different evaluation methods for module graphs is to enhance the scalability and the accuracy of the overall analysis by providing for an evaluation that can accord the subsystems peculiarities. In that sense, both approaches imply individual use cases. For certain system structures a numerical evaluation can be more beneficial than an analytical and vice versa. Thereby, it is conceivable to use them in combination for assessing large systems with multiple modules. Section 2.5 gave a general overview on the elemental assessment paradigms and pointed out some essential characteristics. The next section connects on these points by discussing universal attributes for favouring a simulation-based, over an analytical approach and vice versa, with respect to choosing the module's evaluation method.

5.2.1 Choosing the Approach

If we have no limitations regarding the existence of a mathematical description of the system that would deny an analytical evaluation, the deciding factors for choosing the evaluation approach of the module are weighted between complexity of the model, accuracy of the results and required evaluation time. Simulation-based approaches generally require more effort, since the models can cover complex system behaviours, the evaluation itself is very time consuming and the required number of iterations to receive a reasonable result cannot be determined beforehand. Thereby, the module size is not the deciding factor and even the evaluation of small systems may require a large number of iterations of the simulation to receive realistic and accurate results, especially when very low failure probabilities lead to a late entry of the fail state. As mentioned before (see Section 2.5.2), weighted

MCS could be an option in these cases. The required simulation time is related to the modelled failure probabilities and the desired mission time. Substantially, the higher the failure probabilities of the mandatory system components, the lower the required simulation time, as the termination criteria can be reached faster. However, then the amount of iterations to obtain satisfying results may have to be increased. Yet, in contrast to analytical evaluation methods, a result will always be produced without the concern of encountering state space explosions. Principally, analytical evaluation is very fast but suffers from an immensely growing state space, because, in contrast to simulation, every possible state must be constituted before the analysis is performed. As we could see earlier, this state growth is usually exponential with the amount of components impacting the evaluation effort. Thereby, uncritical components that do not constitute to reaching the fail state (components that are not part of the mode of operation) are inflating the state space without contributing to the termination criteria. Though, other than for simulation, the failure probability of components and the resulting probability for reaching the fail state are negligible. Thus, generally, it is more beneficial to evaluate large and complex systems numerically by simulation, while small systems would potentially create an unreasonable overhead and can be assessed analytically with rather low effort. Further it makes sense to model components of lower complexity analytically under the condition that established failure (and attack rates) already exist, e.g., provided by the manufacturer.

5.2.2 Recursive Analytical Approach

The idea of the recursive analytical approach is to reduce the state space of an original system by splitting the evaluation up to multiple tasks of Markov analysis through modularization. An hierarchical dependency exists, as the abstracted graph acts as a superordinate system with one or multiple subordinate Markov analyses.

As pointed out previously (see Section 4.2.2), the original graph's state space grows by $3^{|\mathcal{N}|}$ in the worst case. By the creation of a single module we can split this state space in the following way: The module graph now only consists of the partitioned nodes and thus its state space is described by $3^{|\mathcal{N}_{sub}|}$ in the worst case. The abstracted graph contains all remaining nodes plus the node that represents the module n_{sub}, yielding $3^{|\mathcal{N}|-|\mathcal{N}_{sub}|+1}$ states in the worst case. As an example, consider an original system that consists of ten nodes: $|\mathcal{N}| = 10$ and a corresponding modularization with $|\mathcal{N}_{sub} = 5|$. The original system yields $3^{10} = 59049$ states, the module graph transforms to $3^5 = 243$ states and the abstracted graph

to $3^6 = 972$ states, enabling a state space reduction of $59049 - 1215 = 57834$ states.

Figure 5.4 visualizes the procedure of the hierarchical evaluation with one module and the goal of analysing the overall failure probability of the system. Starting

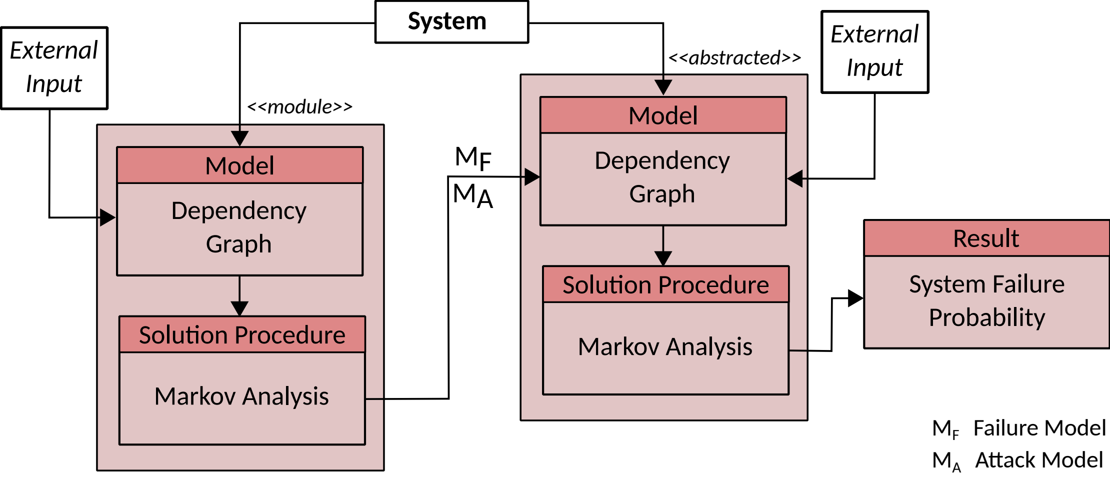


Figure 5.4: Analytical Evaluation Procedure in a Modularized System

with an overall description of the system with respect to the safety and security dependencies of its components, an abstracted and a module dependency graph are built in accordance to the defined formalism. As external input the dependency graphs are provided with failure and attack rates of the individual system components, given as before by, e.g., the component's manufacturer. To analyse the graphs via probabilistic Markov analyses they are both transformed into their corresponding CTMCs (the DMC).

The CTMC of the abstracted graph is essentially the original graph's CTMC without the states of the module nodes $\mathcal{N}_{sub}$ and therefore is extended by the states of the single module node n_{sub}. Its transition rate matrix contains the same rates as before with one exception: The rate for transitioning the nodes replaced by the module node from ok to corr or def (while keeping all others identical) needs to be determined through an evaluation of the module graph's CTMC. It is important to note that this rate actually consists of two rates: First, the rate to transition into the module itself and finally, the rate it takes the module to transition to a corr or def state. In the previous modularization example of Figure 5.1, the rate to transition into the module would simply be the state change of node n_7 in the abstracted graph. The rate it takes the module to transition to corr (in the security case), would be the state change of n_3, which requires the prior corruption of n_1 and n_2. Hence consecutive dependencies exist. To handle

these, we make use of the Chapman-Kolmogorov-Theorem and the fact that there is only one path for this transition. The Chapman-Kolmogorov-Theorem states that the probability to reach a state j after time (t+s) is equal to the product of reaching state k in time t and then reaching state j in time s, summed over all states k that lie on a path from the current state to state j (for more information see [Pap84]). Concluding, the second rate has to be parametrized by time. That is, the higher the point in time t the CTMC of the abstracted graph is evaluated at, the higher the rate for the CTMC of the module graph needs to be. This can easily be explained by the idea that the more time has passed once the first node of the module has changed from ok to corr, the more likely it is that a mandatory node (a node that is part of ϕ_m) in the module has changed to corr. Hence the obtained probability is not described by a steady function and the corresponding rates cannot be calculated sufficiently based on a single probability determined for some specific point in time. This means that an individual evaluation for every desired time point has to be performed.

To make use of the obtained probability (the failure and attack models) of the module in the abstracted graph's CTMC, it has to be transformed into a time-parametrized rate, as mentioned above. This can be done via the relationship $p_{s,s'}(t) = 1 - e^{-R(s,s')\cdot t}$, where $p_{s,s'}(t)$ denotes the probability that the transitioning from s to s' has happened before time t, and $R(s, s')$ is the transitioning rate from s to s'. That is, given the probability for time t from the module, the rate can be calculated as

$$R(s, s')(t) = -\frac{\ln(1 - p_{s,s'}(t))}{t} \tag{5.2}$$

The pictured procedure must be performed for every evaluation step. Meaning, if the property is meant to be evaluated for 12 months of system operation, the process must be performed 12 times.

5.2.3 Hybrid Approach

Similar to the analytical approach the state space of the system is reduced by splitting the evaluation into multiple tasks. Thereby the evaluation of the abstracted graph remains unchanged, implying the same state growth of $3^{|\mathcal{N}|-|\mathcal{N}_{sub}|+1}$. For the module, on the other hand, the state growth is immensely decreased by making use of a simulation-based evaluation, here focused on the Monte Carlo technique (see also 2.5.2), that relies on a different kind of modelling. Simulation models have the advantage of describing the system behaviour, in an event-based manner. Thereby, the internal system states (on computation level) depend on the evaluation goal. For instance, if we are interested in analysing how many times

or to what probability a component turns defective, we must model a state that describes it. For our focused evaluation goal this means we must model a fail state that expresses that the system is not capable of providing its core functionality (as before). The transitions which lead into that state are defined by probability/rate induced events, which result from the dependency graph structure. These events correlate to the different paths of the Markov chain. The key benefit of Monte Carlo Simulation over Markov analysis is that it is more flexible and since we are not bound on satisfying the Markov property, we can easily abstract and combine system states. However, building the event-transitions can become very complex. In that manner, the application of simulation has the advantage of never encountering problems with the state space in the module and thus allowing for the evaluation of very complex module graphs.

Conceptually, the idea of this hybrid approach is to compensate for the disadvantages of one method with the benefits of the other. For example, by applying the simulation model on highly complex subsystems with low failure rates, while the abstracted graph concerns more simple structures with perhaps a complex but static redundancy definition. With that we can achieve a solution procedure for the original system that can be applied in a reasonable amount of time without a heavy penalty on the result precision. While this is possible in theory, in practice it requires a sophisticated balancing between the models that considers the advantages and disadvantages of either evaluation approach. Otherwise the benefits of the hybrid may not be exploited. In the end, the module must cover *enough* components, so that the abstracted graph, evaluated by Markov analysis, does not end up in trouble with the state space. At the same time, these components must have been subject to an appropriate modularization and the yielding subsystem must be suitable for simulation.

Figure 5.5 visualizes the procedure of the hybrid which is equivalent to the procedure of the analytical approach, except for the module's evaluation. Based on a safety and security dependency focused system description and the failure and attack models of the components, a modularized dependency graph is obtained. The abstracted graph is evaluated by Markov analysis and the module graph is reinterpreted as a probabilistic simulation model acting as a basis for the MCS. As hitherto, the currently evaluated time point in the abstracted graph must match the module and we are obliged to evaluate the module first. Since this simulation is much more time consuming and requires several iteration runs to provide accurate results, it would be unreasonable to analyse the abstracted graph and the module in an alternating way, and instead it would be much more reasonable to determine the failure probability of all time points in question in one simulation process. This is, nonetheless, an implementation detail. Contrary to the

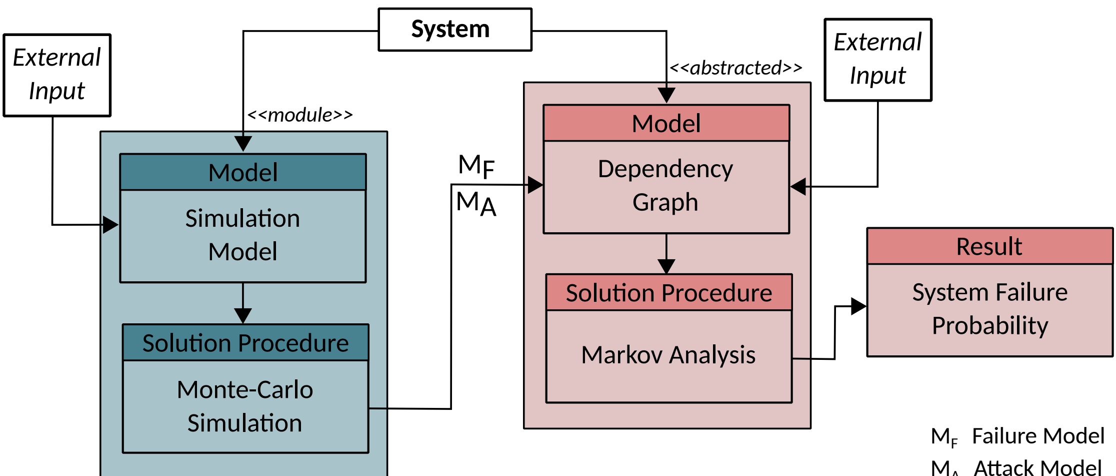


Figure 5.5: Hybrid Evaluation Procedure in a Modularized System

recursive analytical approach, various mathematical distributions can be used to model the failure and attack behaviour. In our case, however, exponentially distributed failure and attack models are assumed. Probability distributions based on these input rates are established for every component. Then, random samples are drawn in each iteration of the simulation for every probability given by the system components failure and attack behaviour, to indicate the time points where the components are simulated to turn defective or corrupted. Similar to the Markov model, the initial state, which is the starting point of each iteration, is the state where all components are healthy (ok) and state transitions are marked by the health status change of a component. This is, however, dependent on the modelled states, because as mentioned, the simulation model abstracts the states we see in the Markov.

An iteration of the simulation finishes as soon as a pre-defined point in time is reached. In our case this is the maximum time point that shall be evaluated in the abstracted graph. Afterwards, the state gets reset to the initial state and the simulation is repeated by initializing a new iteration with a new set of random samples. In every iteration the system state of the evaluated time point is stored for later analysis (e.g. did we reach the fail state?). The number of performed iterations must be defined uniquely in consideration to the individual characteristics of the system, which can be very difficult to determine. In general it applies that the higher the number of iterations the more accurate the results. When the simulation is finished, the system states of the sampled time points are viewed and an average failure probability can be determined. For example, if our goal is to analyse the failure probability of the system in every month during 12 months of continuous operation, like in the previous example, the modelled system state

in every month is saved with the 12th month marking the termination criteria of each iteration. Then, for each time point the number of times the fail state was reached (in other words, where ϕ was not satisfied) is counted and the average probability is calculated in regard to the number of performed iterations. Lets say 100 iterations were performed and in 10 of these the fail state has been reached at time point 12, then the failure probability of the system accords 10% after 12 months of continuous operation.

In order to feed the abstracted graph with this data it is required to reformulate each obtained failure and attack probability in terms of rate. This can be done by rewriting the probability density function similar to the evaluation of the Markov model:

$$r_t = -\frac{\ln(1-p_t)}{t} \tag{5.3}$$

with t being the time point, p_t the obtained probability at that time point and r_t the according rate.

Hybrid models and hybrid modelling can be categorized by different classes according to Sargent [Sar94]. Following this classification, the presented hybrid approach categorizes as *class III*, because the numerical evaluation (simulation) is applied on a subordinate level. In [RH20a] the hybrid approach was firstly presented with the goal of supporting the design of safe and secure autonomous vehicles, by providing an approach that is able to analyse large and complex systems and its reliability parameters quickly, precisely, realistically and easily.

5.3 Heuristics

The last sections presented the formalism of modularizing dependency graphs and showed the evaluation possibilities. Thereby, we could already see that certain graph structures forbid a modularization. In this section, the accuracy of the modularization is discussed and structural aspects that affect the quality of the modularization are pointed out. In the end, a loosely defined rule set based on observations throughout the application of the modularization approach is established.

5.3.1 Accuracy

Naturally abstraction causes loss of information, meaning the retrieved results can by definition not be as exact as compared to an equivalent non-modularized system

and a compromise between finding a scalable and solvable system and the precision of the result must be made. As we saw earlier, the modularization is capable of drastically decreasing the required state space in the solution procedure, even if the purely analytical approach is used. This is an immense benefit for the scalability, but what does it mean for the result accuracy? Here some challenges due to the characteristics of the methodology must be faced. Firstly, without modularization each state in the Markov model is representing a health state of the graph and thus valid paths of the original system. When we slice that graph up into a module and an abstracted graph, yielding two (more or less) individually solved systems, failures and attacks that alternate between nodes of the abstracted and the module graph are obscured. Figure 5.6 shows an example for this based on the system

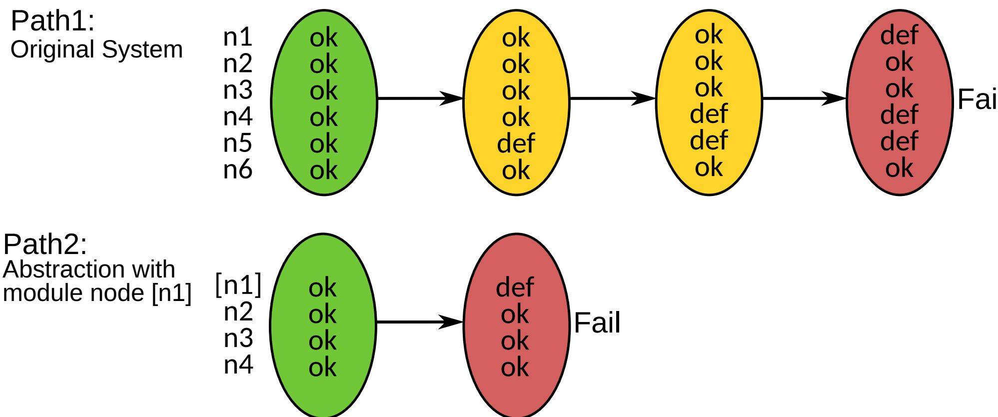


Figure 5.6: Example Path through Modularization

previously introduced in Figure 5.2. The upper path belongs to the original system and the lower path is its equivalent in the modularized system (abstracted graph). Due to the previously defined modes of operation, the system failure is reached as soon as n_1 or n_2 turn defective as a result of an internal failure or the failure of data providing nodes. When comparing both paths, it can be observed that no single failures in the module can be considered any more, as a failure of the module already results in a failure of the entire system. In the abstracted graph it cannot be distinguished whether a redundant or a mandatory component failed in the module. Thus the failure rate evaluated for the module must approximate this behaviour. Although this may cause an issue when evaluating the original and the modularized system, it also shows the benefits for the scalability of the approach.

Secondly, the method was explicitly designed so that the model holds intertwined safety and security effects, though now we are required to determine independent safety and security rates for the module. Precisely, one rate that describes that the

module turns non-operational on behalf of one or multiple safety failure(s) and one that describes its inoperability due to one or multiple corruption(s). But since both effects may be present at the same time, and perhaps even alternating, how can the degradation cause be narrowed down? For instance, a system may experience several non-critical safety failures where there still exist a redundant component to take over the task. In the end, the transition to the Fail state happens due to a corruption of a critical component (see the exemplary path in Figure 5.7). Did the system fail safety- or security-wise? In addition to that, corrupted nodes may turn

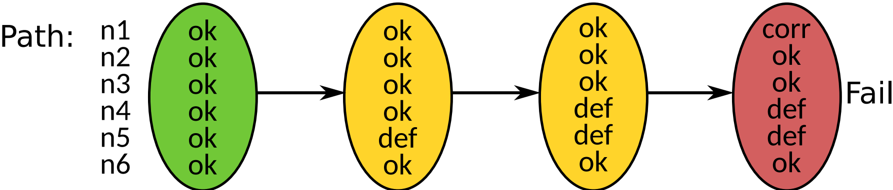


Figure 5.7: Imaginary Path with Defects and Corruptions

defective, leaving the represented component useless for the attacker, however, for the system the component was non-operational already. To avoid this problem, two distinct modules, one for safety and one for security, could be established and evaluated separately. However, then the core concept of the method is neglected and intertwined effects completely ignored, for instance, a functional dependency being violated due to the corruption of a supporting node. This yields a different methodology that does not accord with the original system and will lead to a result divergence.

Since it does not make sense to change the methodology to solve this problem in its detail, an approximation through the evaluation property has to be made. Several ideas are conceivable. In the simplest way, we just view the state change of the transition into the Fail state; if it was a node turning corrupted, it counts as a security failure (just like in the previously displayed path) and if the node turned defective, it counts as a safety failure. Though this has some downsides; the property with the higher rate for occurrence is potentially shadowing the probability of occurrence for the other. As an example, if the failure rate is high, the node may turn defective before it has the chance of being occupied by an attacker and turning corrupted. Furthermore, this neglects the cause of previous degradations which made this transition to the Fail state possible. To include these, we could take the number of defective and corrupted components into account. For example, in paths where the fail state is reached and there exist more (critical and redundant) defective components than corrupted, it counts a safety-wise failure and vice versa. Following that idea, the path in Figure 5.7 would count as a safety failure. Yet, this is problematic because oftentimes not all components are reachable and thus

can be corrupted. This would most-likely neglect many corruption-based failures, increasing the safety failure rate while decreasing the security failure rate. Since this is not an option either, this thesis is focused on the cause of reaching the Fail state.

5.3.2 Well-defined Modules

Subsequently, loose rules to build well-defined modularizations based on observations are given. While most of these apply across both analysis approaches by being a result of the methodology, they are based on observations throughout the appliance of the recursive analytical approach and help of the model checker PRISM. Thereby, the shown examples are kept small to remain comprehensive. Of course these would not require a modularization due to exploding state spaces. The PRISM code of the displayed examples can be found in Annex A.1 and the results of all performed computations in Annex B.1.

Resulting from the previous accuracy discussions, in general a higher accordance with the original system can be achieved if the modularization candidate only models one property; safety or security. Though, albeit a non-reachable subsystem is conceivable, a subsystem without safety failures is unrealistic, making this rule difficult to apply.

Further observations showed that the path property of Reach-links, despite being taken care of with the time parametrization in the module, is still difficult to approximate. Consider a system where the module contains a Reach-path as in Figure 5.8 with $\phi = \hat{n}_7 \vee \hat{n}_4$. With n_7 and n_4 being redundant, the Fail state is reached when both nodes turn corrupted (only considering security attacks for simplicity here). In this way, it is made sure that both paths (left and right of n_5) contribute to the system failure. Due to the criticality of n_4, the module node formed in the abstracted graph is also critical and redundant to n_7. Generally,

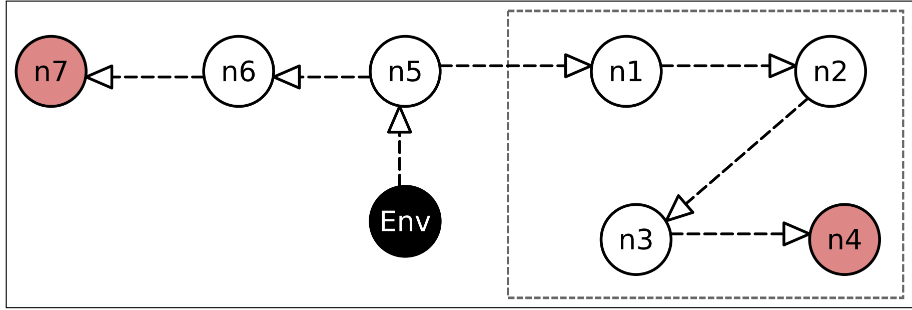


Figure 5.8: Problematic Security (Reach-path) Modularization

Reach-links employ a time dependency on the rate of the concerned node: A node that is not directly reachable may only turn corrupted once a node with a targeting Reach-link is corrupted. With the here pictured path the module node, as well as n_1 in the original system, can only be attacked once n_5 turned corrupted. Though, in the module graph we must model a direct Reach-link to n_1 which indicates that it can be corrupted unconditionally from the start (by the set rate). Even though we take care of the corruption of n_5 in the abstracted graph to happen first, the module's result will be over-approximated.

To show this, the system is run with two different attack rates for n_5 as pictured in Figure 5.9. On the left hand-side, all nodes including n_5 were given an attack rate of 0.2. On the right hand-side, only the attack rate of n_5 was set to 1. The failure functions of the original and the modularized system for attack_-rate(n_5) = 0.2 show a small but clear divergence. However, with the evaluation of later time points they converge again as they naturally strive towards 100%. In comparison to the other plot with a higher attack rate on n_5 we can see that the modularization is more accurate. This is because in that case, the original system more closely resembles the *instantly available* transition we have in the module.

Consequently, modularized systems that contain lengthy Reach-paths pointing to a module should be avoided, or the path should be integrated in the module. Yet, in case the attack rates in the abstracted graph's path are high and respectively low in the module, the impact on the evaluation results of this effect is reduced. Another option (that is however not in the scope of this thesis) would be to determine a correction factor for the approximated module rate based on the length of the Reach-path pointing the module and in consideration of the set attack rates. For example, [ALL18] calculate correction factors to increase the accuracy of simplified Markov models.

Since the corruption of a node describes both, the takeover of the node by an attacker who may abuses it to attack neighbouring nodes, and the implied loss of functionality, another problem arises in the evaluation goal of the module. In the original system we are usually focused on the implications to the system's functionality alone, i.e., when the Fail state is reached as the formula of the mode of operation evaluates to False. This applies to the majority of realistic modules where its failure leads to the failure of the system and/or it does not have outgoing Reach-links. Nevertheless, it is also viable to model a non-critical (or redundant) module with outgoing Reach-links as displayed in Figure 5.10. As an example, imagine $\phi_m = \hat{n}_1 \vee \hat{n}_5$ so that the system's Fail state is reached if both n_1 and n_5 are corrupted (for clarity, defects are again not considered). Hence the module has a redundancy relation to n_5. When evaluating this kind of module we actually

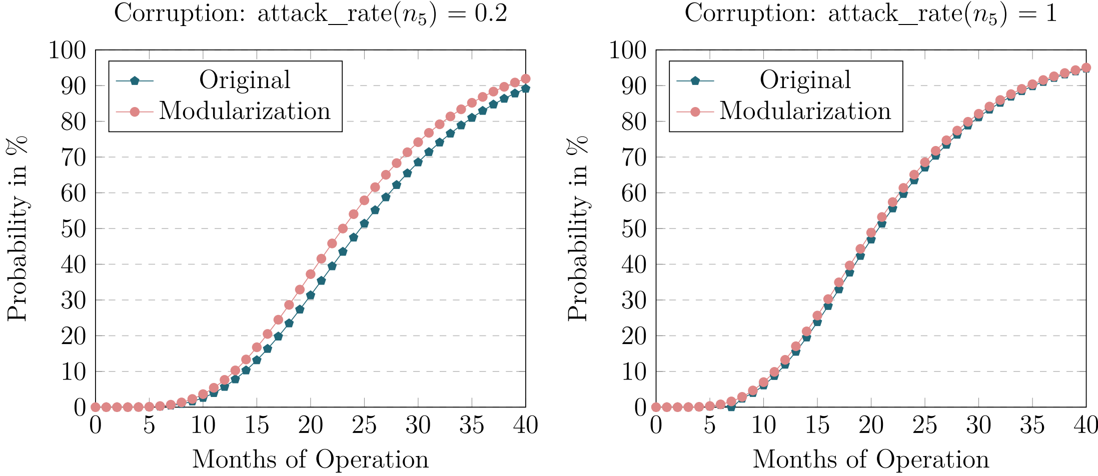


Figure 5.9: Plot: Problematic Reach-path Modularization

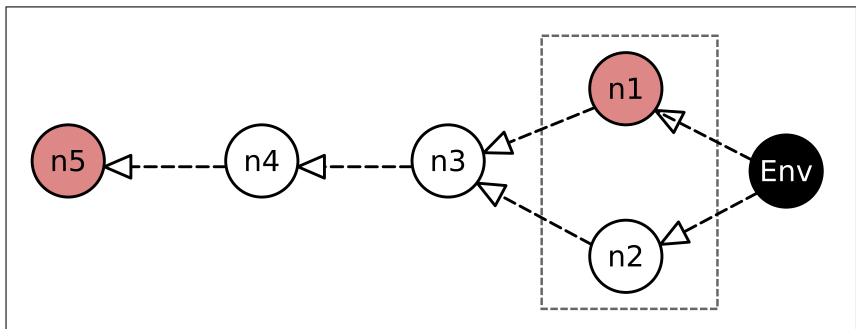


Figure 5.10: Problematic Security (Reach-link) Modularization

require rates for both; entering the Fail state to determine that the module node part of ϕ_a has failed due to a corruption, and the takeover by the attacker to express that the node can serve as a platform to attack further nodes. The first rate can be determined by using ϕ_m. The latter is given by all nodes that have outgoing Reach-links to nodes in the abstracted graph, which allow the attacker to corrupt nodes outside the module from a node inside (here n_3). However, it is not possible to determine two distinct rates for the corruption of the module. If we follow the idea via the mode of operation as before, since n_2 has a Reach-link to a node outside the module but is not required for the requested functionality of it (specified by ϕ_m), the possible impact of the attacker residing in n_2 would be abstracted away by the modularization. But if we change the dependency graph so that n_2 is also part of ϕ (and thus ϕ_m), its attack rate will be taken into account and the result precision increases. Thereby, we have to be careful to correctly

picture the dependency of the main module: An appropriate mode for the module would be given by $\phi_m = \hat{n}_1 \vee \hat{n}_2$. It is essential that both nodes are seen as redundant by the use of the logical or operator in order to express that either corruption leads to a subsystem corruption, depicting the corruption of node n_3 in the original system.

To show the impact of this abstraction, an exemplary evaluation with an attack rate of 0.15 for each node was run for all discussed system variations as displayed in the plots of Figure 5.11. The black line shows the original example, the orange one the simple modularization and the red one the adapted modularization, where the path of n_2 was included (the blue line is discussed later on). We can see that the adapted modularization resembles more closely the original system, yet does not match it perfectly. Consequently, for these kinds of modules, a set of nodes with a non-trivial ϕ_m is not suited for modularization if there is some node that does not occur in ϕ_m but has a Reach-link to some node outside the set.

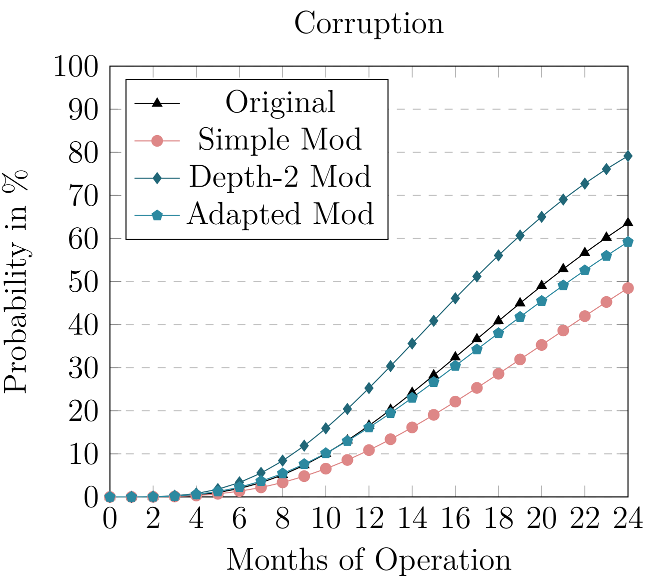


Figure 5.11: Plot: Problematic Reach-link Modularization

Despite that, a possibility for avoiding this problem while still realizing a modularization of the original system exists. Therefore, we must extend the recursion depth and define a module inside a module so that nodes n_3, n_4 and n_5 form a module with depth 2 (hence the submodule) and nodes n_1 and n_2 with depth 1, pointing to it via a Reach-link (see Figure 5.12). This leaves the abstracted graph simply consisting of a single node. In this specific example the abstracted graph

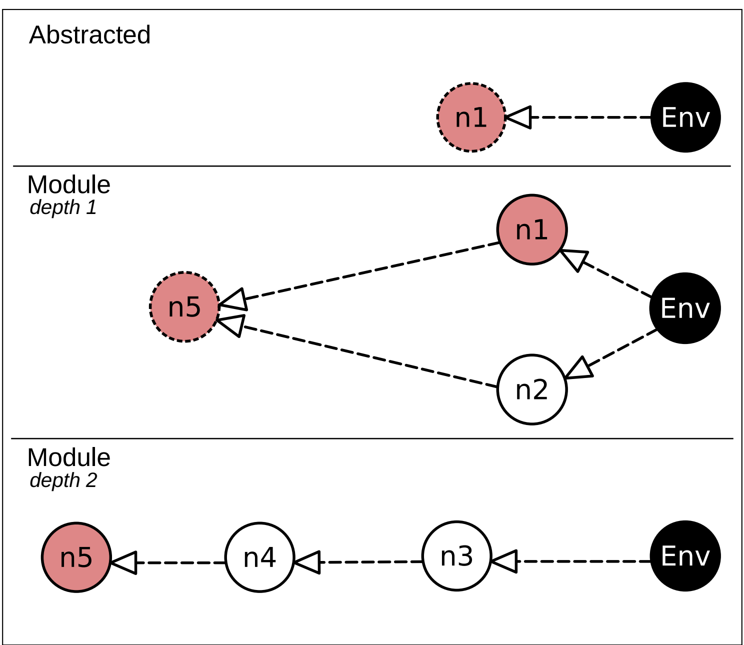


Figure 5.12: Depth-2 Modularization

could be omitted, because it would not change the semantic, yet it slightly affects the results because a transition is added. While in this way a modularization becomes possible throughout maintaining the path properties of n_1 and n_2 as before, we run into the same problem as discussed earlier; now the submodule (depth 2) has a Reach$-$*link* from Env, indicating an instantly available transition which does not match the original system. As a result, an over-approximation is made as depicted by the blue plotted line in Figure 5.11. In special cases this may become an option, e.g., same as before, when attack rates in the module are low. Though, it must be mentioned that a higher recursion depth complicates the modularization as it increases the number of module nodes and the thereby required evaluation runs which cannot be parallelized.

Furthermore, the security mechanism provided by one node to another cannot be transferred to the module node, as its guarantee is substituted from the intrusion rate of the target node in the evaluation as long as this node is functional. Thus the guarantees provided by the security mechanism are part of the transition rate into the corrupted state of a node, which makes it impossible to calculate a detached rate for provided security mechanisms of a module in the present model. Given that, modularizing systems where single nodes have Sec-links beyond the border of the module should be avoided, as they are prone to leading to inaccurate evaluation results. Depending on the system, it could be an option to simply transfer the

provided security guarantee to the module node, though, this can only be an option if there is not more than one module node providing security guarantees and also this is a hard over-approximation.

6 Recovery

With the omission of the human control instance autonomous vehicles are required to reach a certain degree of self-awareness. Meaning, the vehicle must possess knowledge of its own state, possible actions and the result of these action for itself and the environment [Sch+17]. This is an absolute requirement in order to act appropriately and safely in any conceivable scenario. For instance, the vehicle must be able to identify whether it is driving in the city or on the highway/autobahn and it must recognize special situations such as accidents and act accordingly. Additionally, self-awareness on a functional level is the key to becoming fail-operational, i.e., capable of tolerating single failures at least until a safe stop is reached [Sch+17] (see also the previous Section 2.4). Thereby, the system monitors its functions and identifies degradations or misbehaviours. In order to counteract, it must know about redundant modules and activate them if in need. Self-awareness and especially automatic recovery, referred to as self-healing, are properties connected to the field Organic Computing (OC).

Generally OC includes concepts that aim to make currently strict and inflexible technical systems automatically adapt to new situations and environments as a reaction to external and internal events, by implementing or rather imitating biological concepts (following the definition by Tomforde [TSM17]). Apart from self-healing it concerns properties such as self-adaption, self-protecting and self-configuration. These are closely related to the recovery of autonomous systems, because they describe highly advanced concepts to overcome flawed system states by, among others, adaptively rearranging communication structures and functional dependencies. Given that, OC concepts offer a promising solution to overcome the challenge of specifying and verifying the complex autonomous system at design-time. Therewith, it can be speculated that in the future, next to classical manual recovery based on diagnosis and logging data, a growing part of a vehicle's recovery mechanisms will be OC-based. However, it must be mentioned that these concepts also may increase the risk through uncertainty; how can we be sure that the system will execute an appropriate healing mechanism at an appropriate time and situation?

Since the potential to recover from flawed states can significantly increase the system's safety and security KPIs, a holistic system assessment must be capable

of reflecting these. Therefore, this chapter aims to derive the nature of different recovery mechanisms with the goal of establishing generalized strategies that can be transferred to the model in a formal manner. Therefore, the upcoming Section 6.1 provides a wide-spread overview on existing recovery mechanisms throughout the hardware level, application level and system level. Based on that, Section 6.2 attempts to generalize these different recovery mechanisms to a computable modelling. Finally, Section 6.3 shows how the specified formalism of Chapter 4 must be adapted to include the identified recovery strategies. The basis for this chapter has been published in [HR23].

6.1 Recovery Approaches

Dictated by the level and area of application, over the years, many different methods and techniques for recovering a component's flawed state have been developed. Thereby, it is only natural that various terms have been established such as recovery, repair or healing. Although no uniform definition exists and literature commonly treats these terms as synonyms, reparation usually indicates a replacement of the concerned faulty part and recovery refers to a restoration of it [Mon18]. In the course of this work the term recovery is used to express any action that is capable of changing a component or subsystem from a former flawed state back into a functional one. However, since we have a static view on the modelled components and view the system in continuous operation, it is disregarded from maintenance actions and repairs that afford the physical exchange of a component or part. Given the fact that safety issues and security incidents can have very different causes, recovery approaches usually are targetted on either one of them. Thereby, security-oriented methods are usually aiming to proactively prevent attacks in first place. In case an attack has occurred regardless of the implemented protection measures, these methods commonly couple the transition back into an acceptable state with further countermeasures to close the exploited security gap, or to stop the attack from spreading. For our context it is assumed that the effect of these kind of security-hardening actions is integrated in the components attack probability or expressed by received security guarantees. Naturally, a recovery action cannot simply happen but is bound to several conditions that are unique to the implemented measure. While some methods require human input for their decisions, self-aware and automatic approaches are on the rise, fitting the context of autonomous systems especially well. To determine and abstract these possible pre-conditions with the goal of formulating them in our model, the subsequent sections briefly view some existing approaches separated by their implementation level.

6.1.1 Hardware Level

Classical hardware level approaches are based on voter systems, such as Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR). Like indicated by their name, they implement redundant modules and let a voter monitor their computed results. In case of result divergence, the voter decides which result to trust. Note that also some software-based approaches rely on this principle such as n-version programming. However, also more revolutionary techniques have been developed. For instance, Khalil et al. [Kha+19] introduce, among others, Evolvable Hardware (EHW) and Embryonic Hardware (EmHW). EHW combines reconfigurable hardware, like an Field Programmable Gate Array (FPGA), with a bio-inspired evolutionary algorithm that can also be used for circuit design and synthesis. In case a fault or degradation is detected, the hardware chip is reconfigured to a new design that excludes the faulty hardware cells. This makes a dynamical adaption possible and increases the lifetime of the chip (availability). EmHW follows a similar concept, though here the reconfiguration takes place on behalf of the hardware cell. Inspired by biological stem cells, each cell is embodying the same functionalities. Moreover, each cell has knowledge of the current functionality of neighbouring cells. In case an active cell ceases to operate correctly, an inactive, spare cell takes over and replaces the faulty cell. Due to that this concept requires very high memory per cell and is still mostly a research topic. In [OT00] an implementation in a commercial FPGA is presented.

6.1.2 Application Level

Given the versatility of software-errors, various concepts for performing software recoveries exist. Monperrus [Mon18] provides an exhaustive overview on *automatic repair*, hence, *the transformation of an unacceptable behaviour of a program execution into an acceptable one according to a specification*. Thereby, the term repair includes other terms such as healing, patching, fixing, correction etc. The paper separates between two categories: behavioural repair and state repair. Behavioural repair describes approaches that change the behaviour of the program, hence its source code or binaries. In online systems such as our context these repairs are essentially dynamic software updates. For example, approaches that revert the software code to an older version ([LR16]) or the repair of crashing exceptions through automatic patching [ANM14]. State repair concerns approaches that are changing the program state, more precisely, its heap, stack, input or environment. Classical state repair comprises the restart (reboot) of the affected component to

reclaim stale resources, clean up the corrupted state and fix Heisenbugs[1] (see e.g. [CF01]). Further, it concerns the rollback to a previous operational state by making use of checkpoints and snapshots, or the entire switch to a different program, preferably written in another language, such as the idea of n-version programming. These presented approaches were mainly focused on recovering from safety based failures. Some, especially behavioural repair approaches, can also work well for recovering from security attacks (malicious code injections). However, state-based approaches like a simple reboot will have little to no effect on security attacks: while Heisenbugs may be extinguished, malicious code will persist. To cover these, attack-dedicated solutions must be applied, for example, DIRA by [SC05] automatically detects and recovers (control-hijacking) buffer overflow attacks by preventing the identified attack from propagating and rolling back to a safe state if necessary.

6.1.3 System Level

On system level we regard approaches that go beyond the hard- or software of a single component, focussing on the identification and correction of poorly performing processes. Oftentimes these approaches achieve their recovery through (self-)configuration or (self-)management. For instance, in the SafeAdapt project [SDW14], a *Safe Adaption Platform Core* was developed. This platform core combines different hardware platforms via an adaptive network structure that lets the resulting system reconfigure itself by intelligently reassigning computational resources. It is claimed that in this way an effective fail-operational behaviour for the architecture of autonomous vehicles can be achieved. Kain et al. [Kai+20] followed a similar idea for achieving a fail-operational autonomous vehicle with FDIRO, an extension of the well-established Fault Detection Isolation and Recovery (FDIR) of the space domain for automotive, by the addition of an optimization step. Recalling, the core of an autonomous vehicle consists of multiple computing units that run several software applications as presented previously in Section 2.2. On failure detection the concerned computing unit is isolated and it is switched to a redundant instance to ensure the operation of the running applications. In the recovery step, the system configuration is adapted so that safety can be guaranteed. However, this configuration may not be optimal, because, e.g., the redundant unit may only provide a degraded functionality (as it is the case with the emergency computing unit of Section 2.2). Therefore, the optimization step tries to reconfigure the process occupancy. For instance, instead of mirroring all processes on the redundant computing unit, some are switched to other computing units

[1] Bugs that are difficult to reproduce because they often vanish during their investigation

to avoid it overloading. As before, these approaches can also work for the recovery from security incidents. However, if not handled with care regarding security, these might become an entry point for the attacker, e.g., if an infected software process is switched between computing units, enabling the chance of the attack to spread further. A security targeted answer to these are IDS. Though most IDS are focussed on detecting and preventing attacks on the system by various intelligent countermeasures rather than providing active measures to recover from successful attacks (see also [Zha+20]). In [Zha+20] an attack-response framework is presented that builds upon an existing IDS and performs a recovery action upon the detection of a successful attack. The general idea is that in the recovery mode the recovery controller takes over and attempts to bring the system to a safe state based on checkpoints within a given, calculated safety deadline (the time the system should be steered back into the target state or the latest time before it becomes unsafe).

Despite automatic recovery actions, it is further possible to steer the recovery from the outside. Therefore, the information logged by the IDS or FDIR approach is sent to the vehicle owner/manufacturer and a manual repair action such as a software update, reboot or the instruction to go to the workshop could be induced.

6.2 Model Abstraction

Based on the presented approaches and techniques we can observe some general concepts. First of all, the success of the recovery action is not certain, because it depends on a successful detection and identification of the problem source. Following, the employed method must be capable of recovering the identified problem. Given these factors the success can be measured as a probability or rate, fitting our model. Hardware and software level approaches concern only the affected component. Consequently, no implications on dependencies from other components are required to be considered. For application level approaches, however, we must consider that there is some sort of external component running as a watchdog that will trigger a repair action on the affected component. For this action to be successful, there must be, among others, an active non-manipulated connection between these components. Equivalent to that, non-automatic recovery that is based on human intervention requires an active non-manipulated connection to the outside.

Concluding three recovery schemes can be derived:

- Self-performed
  Given the internal recovery mechanisms of the component, the component is able to recover itself indicated by some rate or probability of success.
- Performed by another component
  Given the recovery mechanism of the system, the component is recovered by another component indicated by some rate or probability of success.
- Performed from the outside
  Given the system's detection mechanisms in combination with the configuration options of the component and the capability of human interventions, the components recovery can be triggered and steered from the outside indicated by some rate or probability of success.

Furthermore, we saw that different approaches for safety and security exist. Since an approach like a restart can be well suited for recovering from safety failures (Heisenbugs), however, is not suited for recovering from security attacks like malicious code injections and therefore we must distinguish between their success rate. This flexibility increases the detail of the modelling possibilities: For instance, in reality it is imaginable that a highly critical component or system deploys both kinds of approaches with different maturity levels leading to diverging success rates. Likewise, it is plausible that the safety recovery approach is implemented on the component level, while the security recovery approach works on the system level. By separating between them in the model, we can consider different recovery schemes of independent success for the defect and the corruption of a node. Thereby, the probability rate indicating the success of the recovery action is determined by its taken steps. Regardless of the considered level or the definite approach we can observe that the general process scheme is identical: First, a monitoring process must detect and identify the failure or attack. For many application level approaches, this monitoring process works like an *oracle*; analysing whether the observed behaviour matches the specified behaviour. In the next step, the location of the defect or corruption must be found and its cause diagnosed. Once this information is gathered the system tries to adapt itself (or is reconfigured externally to adapt). Therefore, one or more candidate fixes are generated. In the last step, these candidate fixes are tested and the one offering the most promising target state is deployed. In the simplest case there is only one candidate fix. Figure 6.1 visualizes this scheme in orientation to multiple sources like [PD11] and [Kha+19]. In a fine-grained model the success rate of the recovery action could be determined by defining a success rate for every process step. Efforts for modelling these steps in terms of a random distribution were made in [HR23]. However, for

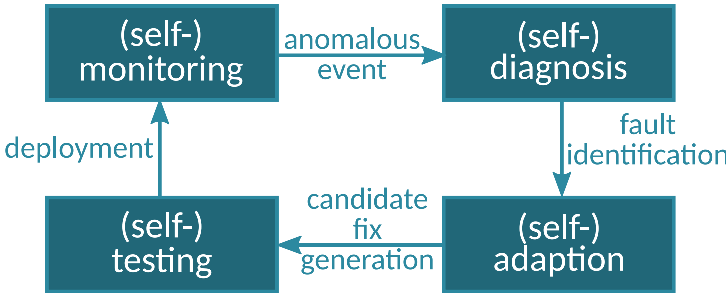


Figure 6.1: General Recovery Scheme

the abstraction layer concerned in the model of this thesis it is merely relevant to what rate or probability the recovery succeeds.

Given this process scheme and according to Khalil et al. [Kha+19], the recoverable system must consist of three states: normal state, degradation state and defective state. In the normal state the system is operating as expected. In the degradation state a fault has occurred but the general operation is maintained. If the system is capable of recovering from this fault, it transitions back into the normal state. Lastly, in the defective state the detected fault resulted in a failure and the operational function cannot be maintained, yet the system can still recover from the failure yielding in a transition back into the normal state. Transferring this to our specification, slight changes have to be made. The normal state is simply the initial state where all components are working correctly. The degraded state is then entered as soon a component has turned defective or corrupted, yet the system is still operational, as the component is either uncritical or redundantly designed. A critical, broken state where the system is still operating and trying to recover does not exist for us. This state is simply the Fail state in that the operation cannot be maintained anymore. Based on the previous contemplations any recovery from this state is excluded as the safety of the system cannot be guaranteed in it, keeping it as before a final state. Thus, a recovery can only take place from a degraded system state back into the normal state (or another *less* degraded state, considering multiple component failures). Due to that, the recovery of a component can enhance the system's availability and reliability. Earlier it was discussed that availability and reliability are equal in systems without maintenance, having the positive effect of simplifying the reliability computation. By excluding recovery from the Fail state, we can see that this equation actually persists: Since we never enter an unreliable state for an arbitrary long period of time to transition back to a reliable state, the system will always be available if it can recover. Accompanying with that, we cannot compute the maintainability of the system.

6.3 Formalization

In regard to the previous considerations, the formalization of the modelling first presented in Chapter 4 must be revisited. Thereby, Definition 4, that is specifying the transformation of the dependency graph to a CTMC, must be modified by extending the transition rate matrix $R(s, s')$. Transitions from the def and the corr back to the ok state must be allowed in one of the determined strategies:

Definition 7 specifies the self-performed strategy.

Definition 7 *$R(s, s')$ is extended by (for $n \in \mathcal{N}$ and $s, s' \in S_{\mathcal{G}}$)*

- $R(s, s[n \leftarrow \mathsf{ok}]) = r_n^{SafeRec}$ *if* $\mu_{s'}(\phi) = true \wedge s(n) = \mathsf{def}$
- $R(s, s[n \leftarrow \mathsf{ok}]) = r_n^{SecRec}$ *if* $\mu_{s'}(\phi) = true \wedge s(n) = \mathsf{corr}$

which reads for the safety case: Whenever node n turned def and the system is still operational there is a probability rate $r_n^{SafeRec}$ with that n transitions back into the ok state and equivalently for security with r_n^{SecRec} from the corr state.

Definition 8 comprises the other two strategies, recovery performed by another component and recovery performed from the outside.

Definition 8 *Let $n_{rec} \in \mathcal{N} \cup \{\mathsf{Env}\}$ be the node that triggers the recovery of n with $n_{rec} \neq n \ \wedge \ s(n_{rec}) = \mathsf{ok} \vee n_{rec} = \mathsf{Env}$.*
$R(s, s')$ is extended by (for $n \in \mathcal{N}$ and $s, s' \in S_{\mathcal{G}}$)

- $R(s, s[n \leftarrow \mathsf{ok}]) = r_n^{SafeRec}$ *if* $\mu_s(\phi) = true \wedge s(n) = \mathsf{def}$
  $\wedge \ \exists \ \langle n_{rec}, n^i \rangle ... \langle n^j, n \rangle \subset \mathcal{L}^{Reach}$ *with* $s(n^i) = \mathsf{ok} ... s(n^j) = \mathsf{ok}$
- $R(s, s[n \leftarrow \mathsf{ok}]) = r_n^{SecRec}$ *if* $\mu_s(\phi) = true \wedge s(n) = \mathsf{corr}$
  $\wedge \ \exists \ \langle n_{rec}, n^i \rangle ... \langle n^j, n \rangle \subset \mathcal{L}^{Reach}$ *with* $s(n^i) = \mathsf{ok} ... s(n^j) = \mathsf{ok}$

For these strategies we require to determine a node n_{rec} that triggers the recovery of n. For recovery performed by another component n_{rec} corresponds to some regular node within the dependency graph that must be operational (ok) for the recovery to take place. For recovery performed from the outside, however, n_{rec} is simply the Env node. Given that, the recovery action can take place under the condition that the command (Reach) path is undisrupted. Hence, there must be at least one Reach-path between both nodes in which all nodes (except for n) are operational. Imagine a watchdog component that is not directly reaching the

concerned defective or corrupted component. To trigger the recovery process, it relies on a communication path via other components to route its messages. The same applies if the recovery is triggered via the outside and the node is not directly reachable from Env. Consequently, this path must not be disturbed in terms of containing any corrupted or defective components.

Naturally the introduction of any new features also rises scalability concerns. Yet, the addition of recovery mechanisms luckily does not extend the state space, because no new states are being introduced. Though, the amount of transitions expands.

6.4 Application and Comparison

Subsequently, the effectiveness of implementing the previously defined recovery strategies is shown. Therefore, a new example system is introduced in Figure 6.2.

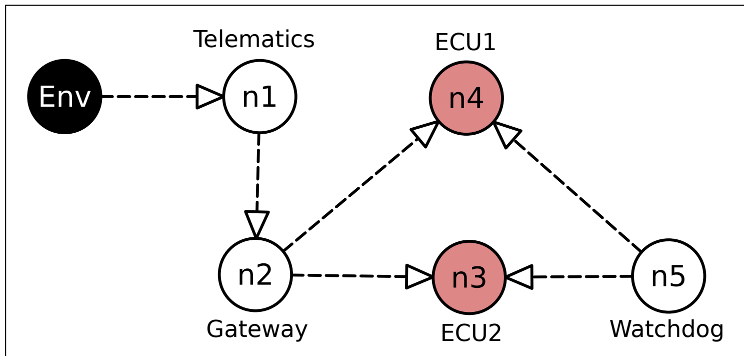


Figure 6.2: Recovery Example System

The centre piece of the system are two critical, redundant ECUs. These ECUs are the recoverable system components. The redundancy is mandatory to show any effect, because otherwise their failure of any kind would lead to the non-recoverable Fail-state. A path to the outside is established via a telematics unit n_1 and a gateway n_2. This path is the only entry point for an attacker, but it is also the only path for externally triggered recovery. The watchdog node n_5 is essentially our n_{rec}, enabling the recovery action triggered by another component.

To show the effect of the different strategies the example is evaluated multiple times:

1. Without any recovery
2. With self-performed safety and security recovery for both nodes
3. With safety and security recovery performed by the watchdog for both nodes
4. With safety and security recovery performed externally for both nodes

Note that for all runs where the watchdog component is not needed it is left without functionality in the graph. For simplicity, all failure, attack and recovery rates are set fixed to 0.05 (which is very unrealistic but helpful to show the effect). The evaluation property for all runs is our prime property, the general system failure, encountered through a combination of safety failures and security attacks. Obviously these four evaluation settings do not exhaustively reflect all possibilities and several other interesting combinations are conceivable, like having multiple recovery paths, variations between failure, attack and recovery rates and so on.

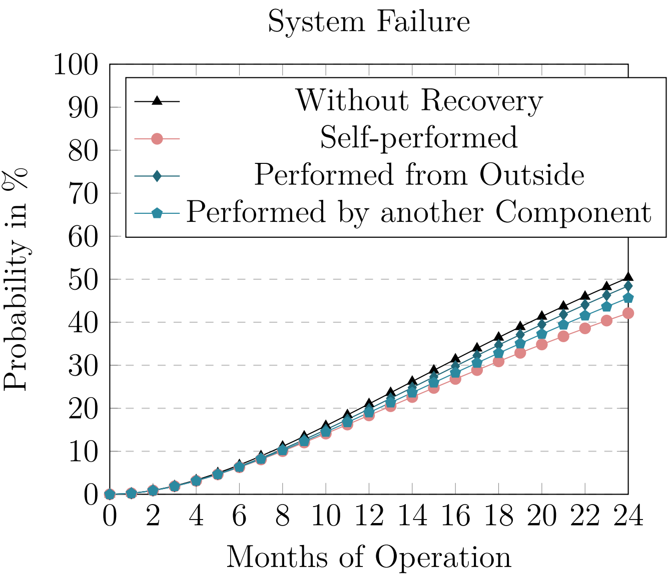


Figure 6.3: Plot: Comparison of Recovery Strategies

The plots in Figure 6.3 visualize the evaluation results. Detailed results can be found in Annex B.2 and the responsible PRISM code in Annex A. Expectedly, the system without any recovery actions fails the earliest. Further, the system with

the self-performed recovery strategy has the best results. This is simply explained by the fact that it has no further conditions restricting the recovery. We can see that the recovery from the outside is performing worse than the one triggered by the watchdog. This is expected as well, since it reflects the system properties: One reason is that the path to the watchdog is shorter and thus its disturbance is more unlikely. Another reason is that the watchdog is not reachable from the Env-node and thus it can never be attacked and turn corrupted. So one *failure mode* is excluded for it. Likewise, the recovery from the outside *shares* the same path with the attacker, making a corruption recovery impossible.

7 Automation: ERIS

While small dependency graphs can still be translated and potentially even solved by hand, realistic examples must cover a plethora of components, and with regard to the exponential state growth of the Markov model, it becomes clear that tool support is needed. To provide a comfortable and efficient way of modelling and assessing dependency graphs, a tool named ERIS was developed. ERIS provides a Graphical User Interface (GUI) for modelling coupled with standard actions such as loading and storing, facilitating the design of dependency graphs greatly. Regarding the analysis, ERIS enables an automatic translation into a corresponding Markov model in accordance with the previously determined formalism. Additionally, a connection to the probabilistic model checker PRISM [KNP11] is established so that ERIS can automatically trigger the Markov evaluation process and prepare the obtained results graphically.

This chapter presents ERIS from its usage to technical internals. It is structured in the following way: Firstly, a technical insight into ERIS and a detailed explanation of its GUI elements is given. This section is meant to serve as a lexicon for the subsequent content. Then, it is described how the formalized model of dependency graphs (see Chapter 4), including the specification of redundancies and the different dependencies, is implemented and performed by the user. Afterwards, the transformation of the basis model into the Markov chain in terms of the PRISM language is discussed. Based on that, the implementation of the previously established extensions regarding dependency graph modularization (Chapter 5) and the consideration of recovery mechanisms (Chapter 6) are portrayed and their implications to the Markov transformation shown. In most parts the specification was followed strongly, though, in some parts modifications had to be made due to the used model checker and also to enhance usability. In Section 7.3 the procedure of both the dependency graph analysis with the recursive analytical approach and with the hybrid approach is examined. The chapter is closed by demonstrating the performance of ERIS on behalf of the analysis of the previously introduced dependency graph of an autonomous vehicle (see Section 4.1.1).

7.1 Technical Insights

ERIS is written in C++ and makes use of the Qt Framework to provide a GUI for a comfortable and user-friendly system modelling. It is available under the GNU Public License (GPL) license on GitHub[1]. Figure 7.1 shows the main window of the GUI. The centre piece is the `Model` tab, where the user can design the

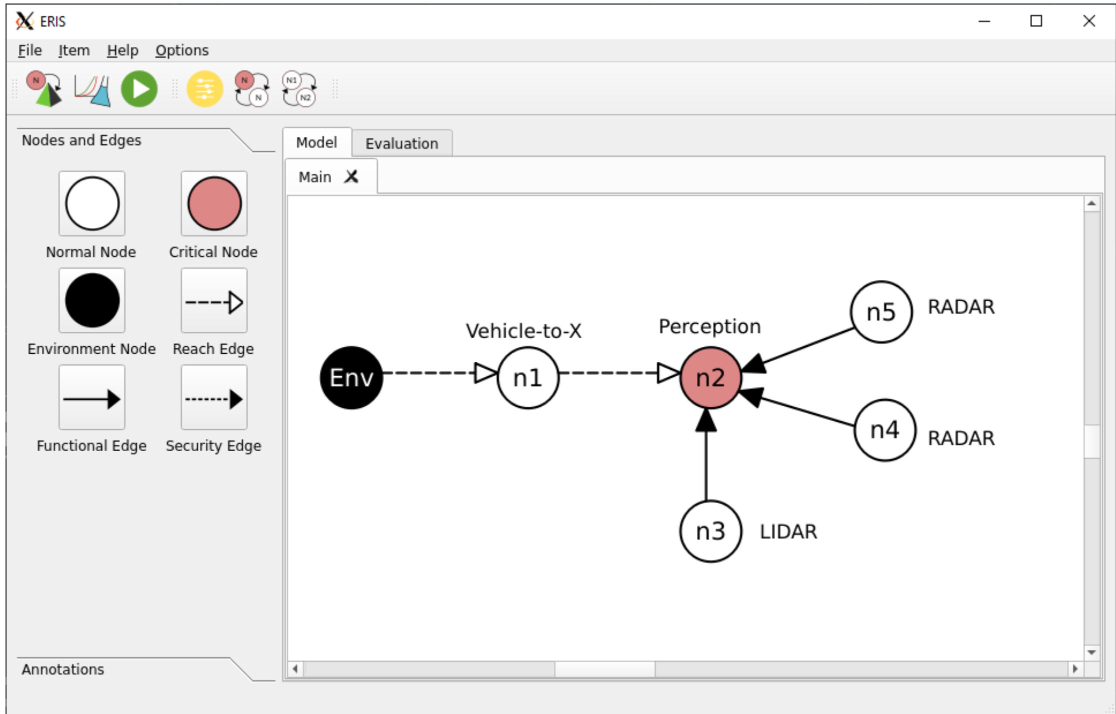


Figure 7.1: ERIS Main Window

dependency graph by making use of the items in the toolboxes on the left hand-side. Thereby two toolboxes are available: `Nodes and Edges` for designing the graph and `Annotations` to create text-based labels as descriptions for the nodes or the system (for more details see Section 7.1.3). Above these toolboxes, the toolbar presented in Section 7.1.2 provides different ways for performing the evaluation as well as manipulating GUI elements. Depending on the performed evaluation process, computed results will be prepared in the `Evaluation` tab. In the top left corner drop down menus are available for file, option settings etc., described thoroughly in Section 7.1.1.

[1] https://github.com/telina/eris/

7.1.1 Menus

Four drop-down menus are placed in the top left corner. The `File` menu contains options for diverse file manipulations.

- `New File` clears the current model tab and resets set file names to default.
- `Save File` stores the dependency graph of the Model tab that is currently in foreground. If no filename has been set by a previous action, the default filename is used and overwritten.
- `Save File as` prompts the user with a file browser to select a file or enter a new filename and then stores the dependency graph of the Model tab that is currently in foreground accordingly.
- `Open File` prompts the user with a file browser to select a file containing a stored dependency graph to be opened in the Model tab.
- `Open PRISM Filename` Prompts the user to provide a filename for the internally generated PRISM file. This option can be useful in cases where the user wants to modify the PRISM file at a different point in time or independent from ERIS. If unspecified, a default name is used.
- `Export Model as` stores the graphical representation of the dependency graph in the Model tab currently in the foreground. The user is prompted to provide a filename and choose a file type. Currently the PNG, PDF and SVG format are supported.

Regarding saving options, dependency graph models are stored in a custom schema of the Extensible Markup Language (XML) format which will be expected from the open action. This allows the user to save and reload the entire model including the individual node settings and the exact placement of all graphic elements.

The `item` menu contains features for manipulating graphic items, at the current time this only includes deleting. The `Help` menu opens a window that provides a short explanation on ERIS' usage. In the `Options` menu, several options regarding the model and the evaluation can be found.

- `Markov Model` allows the user to choose between the type of Markov model that is generated from the dependency graph. Currently CTMCs or Markov Decision Processes (MDPs) are supported. Based on this choice, also the node settings are changed to ask for input rates or probabilities respectively.

- `Interpret Rates as` enables switching the time representation of the provided rates between hours, months and years. This option is only available if the Markov model is set to CTMC, since MDPs are working with probabilities. By default, ERIS expects that all provided rates are hour based ($\times\frac{1}{h}$). The user may use this option item to change the representation of all modelled nodes to month or year and back. Thereby, ERIS attempts to compute the desired representation from the given one for every set rate of each node. Note that for ERIS itself the rate representation does not matter. Thus, this is a feature the user must bear in mind when performing evaluations.
- `Mode of Operation` allows the user to modify the way the mode of operation is generated. While in the default case, `optimized`, the generation accords the formalism where all supporting nodes of critical nodes are taken over (for more details see the upcoming Section 7.2), in the `simple` case only critical nodes (red nodes) are added to the mode of operation. Yet, naturally all transitions where a failure of a supporting node (functional dependency) leads to a failure of a critical node are modelled as before. In this way, the simple mode of operation lets the system fail *later*. Consequently, the amount of generated Markov states and transitions increases and also the evaluation result can diverge slightly.
- `Define Redundancy` lets the user view and define a global redundancy definition over critical nodes by opening an input window.
- `Evaluation Settings` opens a window where the user can modify the evaluation properties used within the experiment evaluation. Figure 7.2 shows this window. Currently the properties `systemfailure`, describing the entry of the Fail-state, `defective`, describing the system's safety-wise failure, and `corrupted`, describing the system's security-wise failure are available. Note that these properties correspond to the automatically generated labels in the transformation process. The user may select which property shall be evaluated within the experiment by making use of the installed checkboxes. Furthermore, it is possible to provide self-written properties. This should, however, be done with care, since the user must know about generated labels or variable names and the property is forwarded to PRISM without semantic and syntactic validation. In addition to the property choice, the user can adjust the time interval and the time steps that shall be evaluated.

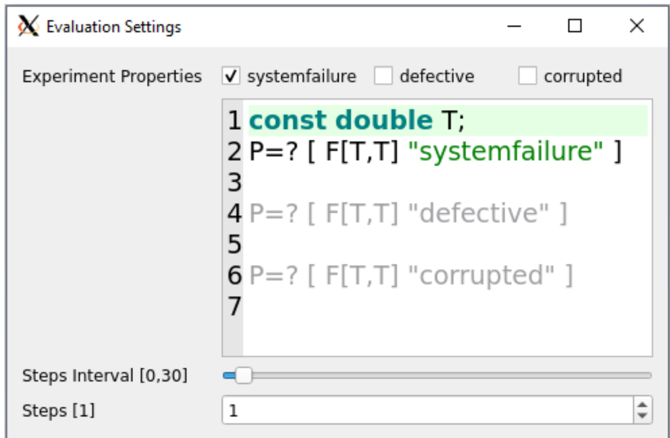
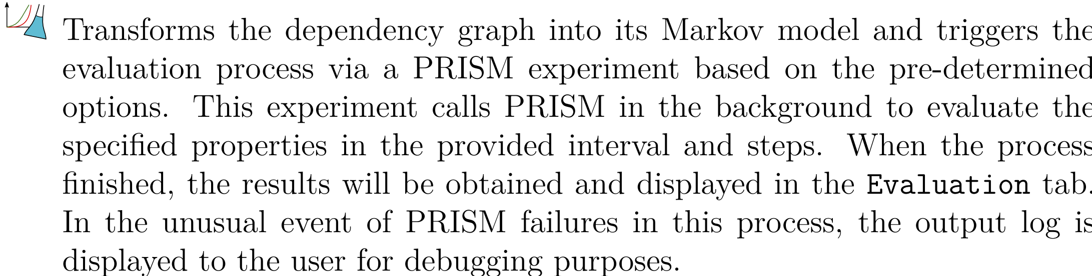


Figure 7.2: Experiment Settings Window

7.1.2 Toolbar

On the left hand-side of the toolbar's separator, actions concerning the evaluation of the modelled system are displayed and, on the right hand-side, options for node item manipulation:

Triggers the transformation process of the currently viewed dependency graph to its corresponding Markov model in terms of the PRISM language. This can be useful in cases where the PRISM file shall be viewed, processed independently from ERIS, or to force a rebuild.



Transforms the dependency graph into its Markov model and triggers the evaluation process via a PRISM experiment based on the pre-determined options. This experiment calls PRISM in the background to evaluate the specified properties in the provided interval and steps. When the process finished, the results will be obtained and displayed in the `Evaluation` tab. In the unusual event of PRISM failures in this process, the output log is displayed to the user for debugging purposes.

Transforms the dependency graph into its Markov model and opens the GUI variant of PRISM for it. This option is useful whenever the user is interested in directly checking the generated PRISM code, or if it is necessary to change it, by, e.g., defining custom labels for the evaluation properties (which is not yet possible in ERIS).

Opens the node settings window for all selected node items (see Section 7.1.4). Alternatively, a double click on a node can be used.

Swaps the criticality of all selected node items. Hence, a critical node will become uncritical and vice versa.

Swaps the ID between **two** selected nodes. Note that this button only becomes clickable when exactly two node items are selected. Alternatively, another ID can be set within the node settings window.

7.1.3 Toolboxes

Two tool boxes are available on the left hand-side next to the graphic scene. The first one, `Nodes and Edges`, concerns all elements required to model the dependency graph. Three node types separate environment nodes, normal nodes and critical nodes. As explained in detail in the upcoming Section 7.2, critical nodes are syntactical sugar for nodes that are directly included in the mode of operation. The different edge types are directly corresponding to the links of the formalism with a `Reach Edge` being drawn with a dashed line, a `Functional Edge` with a solid line and a `Security Edge` with a dotted line. The second toolbox, `Annotations`, offers the possibility of generating a custom text item. These are not connected to any other graphic items and thus can be placed freely in the scene. Further, existing annotations can be modified. Given that, annotations have no purpose in the evaluation but can be used as a descriptive tool, especially meaningful when the graphic scene is exported.

7.1.4 Node Settings

In Figure 7.3 the node settings window is displayed. Reading from top to bottom, in the first two lines the user can select whether the viewed node is actually representing a module or a regular system component. Thereby, two options are available; a Simulation model, where the user is obliged to provide a path to a MATLAB/Octave simulation model, or an ERIS module, where the user must provide the path to a dependency graph file in the XML format. Since the simulation model is not expected to be a dependency graph, unlike the ERIS model, it will not be opened in the GUI. If the node is representing a regular system node, failure and attack rates as well as security guarantees can be specified. Thereby the field `Intrusion Rate` corresponds to the attack rate, `Failure Rate` to the failure rate as before and the `Security Rate` to the security guarantees supplied to another node. An untreated field will be interpreted as zero and the corresponding transition in the Markov chain will be omitted. In other words, a node that has no failure rate can never reach the `def` state. In addition to this basic behaviour

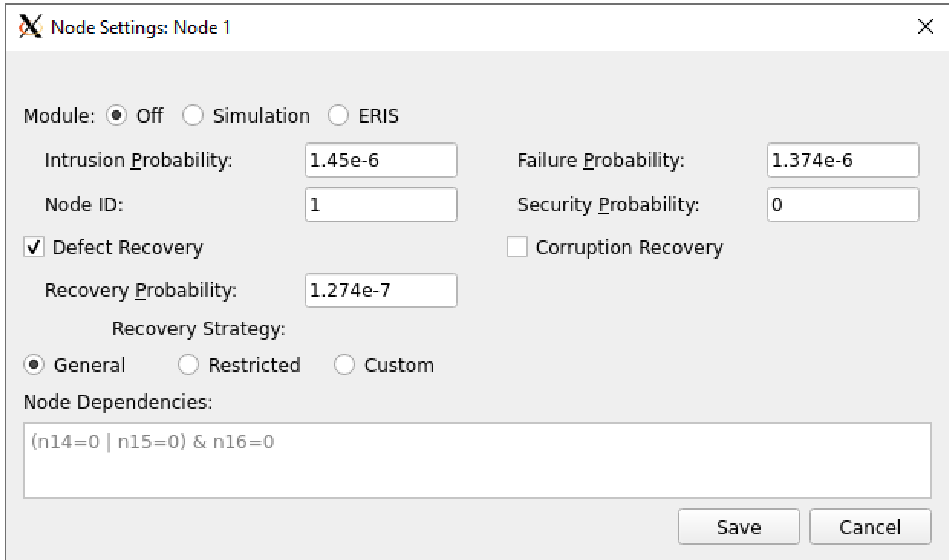


Figure 7.3: ERIS Node Settings Window

the user can define whether the node can recover from a defect or a corruption. Hereby, the three strategies that have previously been defined in Chapter 6 are selectable. Lastly, the field `Node Dependencies` is placed to precisely determine the functional relationships of the concerned node to others. This is required since a Fct-link does not necessarily mean that the provided data is mandatory for the operation of the node, e.g., in cases where a redundant data provider exists, or the link simply shall visualize this relationship. As depicted in the example entry, some nodes n_{14} or n_{15} must be ok and n_{16} must be ok, so that the concerned node can operate correctly. Note that "0" is mapped here to the ok state (for a detailed explanation view Section 7.2.2) and further, instead of our previous symbols "$\vee$" and "$\wedge$" it is made use of "|" and "&".

7.2 Modelling with ERIS

Subsequently, an introduction into the modelling with ERIS with an emphasis on the implementation of the previously defined formalism is given.

7.2.1 Dependency Graph

The dependency graph in ERIS essentially contains the same features as formalized, however, some minor adjustments were made in order to provide a more

user-friendly and straightforward modelling.

The user may choose between three edge types that accord the link types Reach, Fct and Sec of the specification to connect the nodes in the familiar way. Unlike the formalism, there are three node types instead of two available: normal nodes, critical nodes and environment nodes. The environment node (Env) is, identically to the formalism, used to model access from the outside by only allowing outgoing Reach-links. Multiple Env nodes may be declared for visual benefits which is especially useful to keep the dependency graph clean in high complexity architectures with several connections arising from the outside. The differentiation between normal and critical nodes, which are both representing regular system components, is made to make the generation of the mode of operation simpler. Therefore, ERIS internally generates a logical formula `operational` which is later used in the Markov analysis. A normal node is at first a node that is completely uncritical for the system, hence it does not add up to the mode of operation and thus will not be included in `operational`. A critical node, on the other hand, is a node that is mandatory for the correct operation of the system and thus will be added with a logical and (&) relation to the `operational` formula. Yet, indicated by the specified dependencies and redundancy definition, this black-and-white modelling is dispersed: Critical nodes can be defined redundant by using the global (system wide) redundancy definition via the `Options` menu (see Section 7.1.1). Normal nodes, on the contrary, cannot be set redundant for the entire system but with respect to the functional dependencies they deploy. Meaning, in the settings of the target node the `Node Dependencies` must be specified. In this way, the node itself is not viewed as redundant but the data it provides in regard of the individual nodes that receive it. This provides for more elaborate expressions, where, for example, a node is providing two different nodes with data; to one of them it is essential, while to the other it is not. Imagine the system given in Figure

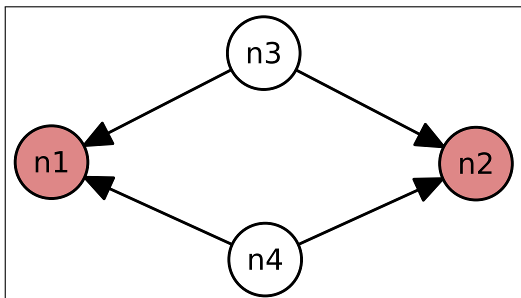


Figure 7.4: Node Dependency Example

7.4, where we set the critical nodes n_1 and n_2 globally redundant, while for the uncritical nodes we specify that they provide redundant data for n_1 but not for

n_2. This yields a mode of operation $\phi = \hat{n}_1 \wedge (\hat{n}_3 \wedge \hat{n}_4) \vee \hat{n}_2 \wedge (\hat{n}_3 \vee \hat{n}_4)$. Note that all nodes that are specified in the `Node Dependencies` must have a Fct-link to the target, making Fct-link in ERIS syntactical sugar. As explained in Section 7.1.4, security guarantees, failure and attack rates are specified in the node settings for each node. With that, all building blocks of the dependency graph are given and the system can be transformed into a Markov model.

7.2.2 Transformation

Whenever any evaluation process in the toolbar (see Section 7.1.2) is triggered, the dependency graph is transformed into its corresponding Markov model in terms of the PRISM language. Note that the evaluation process always concerns the currently active (in the foreground of the Model tab) dependency graph. To avoid overhead through unnecessary rebuilds the transformation is only performed when changes on the graph, options or node settings are recognized. The transform button though (see Section 7.1.2) always forces a rebuild.

The transformation follows the formalization specified in Chapter 4: In the first step, for each node constants of the type double are specified to hold the defined attack rate, failure rate and security guarantees, as exemplarily shown in Listing 7.1 for a node n_1.

```
3  const double rn1SEC = 1.45e-6;
4  const double rn1SAFE = 1.37e-6;
5  const double rn1GUAR = 0;
```

Listing 7.1: Definition of Constants

Note that in this example n_1 is not providing any security guarantees. Then, individual variables that reflect the three health states of the node are declared. Due to the impossibility of defining custom types in PRISM, we simply make use of an Integer variable with a range from 0 to 2 and map the resulting states to our three health states:

$$\begin{aligned} \mathsf{ok} &\mathrel{\hat{=}} 0 \\ \mathsf{def} &\mathrel{\hat{=}} 1 \\ \mathsf{corr} &\mathrel{\hat{=}} 2 \end{aligned}$$

so that a node n_1 in PRISM will be defined as given in Listing 7.2.

```
24  n1: [0..2] init 0;
```

Listing 7.2: Node Definition

Every node will be initialized with 0 to accord the previous assumption that in the beginning every modelled component is ok.

Now the basis for defining transitions is established. Since these depend on the structure of the dependency graph, the creation of all kinds of transitions is shown on behalf of an example system. Therefore, we revisit our previous motivational example of Chapter 4, Figure 4.1 and create a reasonable mode of operation so that in ERIS the computing unit n_1 and the actuators n_3 were marked critical as shown in Figure 7.5. Further, the sensors are providing functionally mandatory

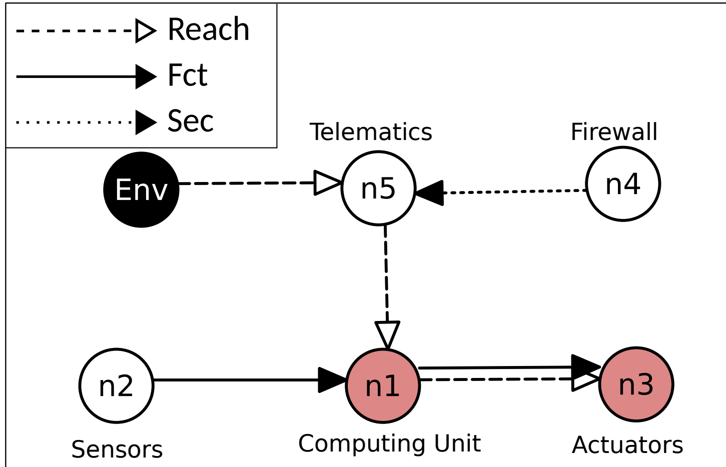


Figure 7.5: Transformation Example System

information to the computing unit. This yields the mode $\phi = \hat{n}_1 \wedge \hat{n}_2 \wedge \hat{n}_3$. ERIS represents this by a formula called `operational` in the PRISM language as shown in Listing 7.3. This formula is used to define the fail state and restrict transitions out of it (lowering the state space).

```
20  formula operational = (n1=0 & (n2=0)) & (n3=0) ;
```

Listing 7.3: Mode of Operation

Supporting nodes are stored in a dedicated formula for every node that receives functionally relevant data (via Fct-dependencies). Listing 7.4 shows these for n_1 in our example:

```
19  formula n1essentials = n2=0;
```

Listing 7.4: Functional Dependencies

This keeps the transitions clearer, especially when multiple functional dependencies lead to lengthy expressions and it is needed to model the recovery transitions as detailed in the upcoming Section 7.2.3.

Several different transitions must be generated. First of all, every node may fail safety-wise to its defined failure rate. Thus, a transition into the def state is generated as presented in Listing 7.5:

```
31  [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
```

Listing 7.5: Transition into Defective State 1

In natural language this reads: *Whenever* n_1 *is ok and the fail state has not yet been reached, there is a transition into a state where* n_1 *is defective to the entry rate rn1SAFE*. Additionally to that, a node may enter the def state if its functionally supporting nodes fail. In the example system this can be the case for node n_1, leading to a transition as captured in Listing 7.6:

```
30  [] (n1=0) & (!n1essentials) & (operational)-> (n1'=1);
```

Listing 7.6: Transition into Defective State 2

making use of the previously defined formula `n1essentials` (which is here just relating to the failure of the sensors). Note that in many cases this transition is never taken, as the Fail-state is already reached, captured by the `operational` formula evaluating to False. For the security case the path property as well as possible security guarantees are taken into account. For the telematics unit in our example no path must be considered (previously corrupted nodes), because it is directly reachable from Env. However, the firewall is providing security guarantees. This results in the following two transitions of Listing 7.7:

```
39  [] (n5=0) & (n4=0) & (operational) -> rn5SEC-rn4GUAR : (n5'=2);
40  [] (n5=0) & (n4!=0) & (operational) -> rn5SEC : (n5'=2);
```

Listing 7.7: Transition into Corrupted State 1

Two transitions are required, since only in cases where the firewall is active, the rate for transitioning into a state where the telematics unit n_5 is corrupted must be subtracted by the provided guarantees. In all other cases the defined corruption rate of the node is used. The computing unit not being directly reachable from Env leads to a transition that considers the state of its targeting nodes – in our case the telematics unit as in the below Listing 7.8:

```
42  [] (n5=2 & n1=0) & (operational) -> rn1SEC : (n1'=2);
```

Listing 7.8: Transition into Corrupted State 2

This reads: *Whenever n_5 is corrupted and n_1 is ok, there is a transition into a state where n_1 is corrupted to the entry rate rn1SEC*. Since corrupted nodes may also turn defective, the following transition of Listing 7.9 is established for every reachable node:

```
32  [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
```

Listing 7.9: Transition from Corrupted into Defective State

Based on these definitions PRISM internally builds a matching Markov chain. By introducing a health variable for every node, we are able to define the system's behaviour node-wise, leaving the work of establishing the actual Markov chain to PRISM. Another possibility would have been to generate all possible states resulting from the input dependency graph beforehand in ERIS and defining the Markov chain state-wise with the PRISM language. This would yield a variable for each state rather than each node. While the latter would have the advantage of creating fine grained evaluation criteria, such as the visit of a specific Markov state, it would also immensely increase the transformation effort. Furthermore, then some sort of labelling is required that maps states to node health state(s) in order to reinterpret the evaluation results for the dependency graph and thus allow for making statements regarding the system's estimated behaviour.

Listing 7.10 shows the entire PRISM code.

```
1   ctmc

3   const double rn1SEC = 1.45e-6;
4   const double rn1SAFE = 1.37e-6;
5   const double rn1GUAR = 0;
6   const double rn2SEC = 0;
7   const double rn2SAFE = 1.21e-6;
8   const double rn2GUAR = 0;
9   const double rn3SEC = 1.09e-6;
10  const double rn3SAFE = 1.02e-6;
11  const double rn3GUAR = 0;
12  const double rn4SEC = 0;
13  const double rn4SAFE = 1.52e-6;
14  const double rn4GUAR = 1.98e-7;
15  const double rn5SEC = 1.04e-6;
16  const double rn5SAFE = 1.20e-6;
17  const double rn5GUAR = 0;

19  formula n1essentials = n2=0;
20  formula operational = (n1=0 & (n2=0)) & (n3=0) ;

22  module generatedScenario
```

```
24  n1: [0..2] init 0;
25  n2: [0..2] init 0;
26  n3: [0..2] init 0;
27  n4: [0..2] init 0;
28  n5: [0..2] init 0;

30  [] (n1=0) & (!n1essentials) & (operational)-> (n1'=1);
31  [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
32  [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
33  [] (n1=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
34  [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
35  [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
36  [] (n3=2) & (operational) -> rn3SAFE : (n3'=1);
37  [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
38  [] (n5=0) & (operational) -> rn5SAFE : (n5'=1);
39  [] (n5=0) & (n4=0) & (operational) -> rn5SEC-rn4GUAR : (n5'=2);
40  [] (n5=0) & (n4!=0) & (operational) -> rn5SEC : (n5'=2);
41  [] (n5=2) & (operational) -> rn5SAFE : (n5'=1);
42  [] (n5=2 & n1=0) & (operational) -> rn1SEC : (n1'=2);

44  endmodule

46  label "systemfailure" = !operational;
47  label "defective" = (n1=1 | (n2=1)) | (n3=1) ;
48  label "corrupted" = n1=2 | n3=2 ;
```

Listing 7.10: ERIS-generated PRISM Code

The last three lines show automatically generated labels meant to be used for evaluation, as presented in detail in the upcoming Section 7.3.

7.2.3 Recovery

In order to support the recovery mechanisms introduced in the previous Chapter 6, ERIS allows us to define multiple recovery strategies, separating as before, between safety failures and security incidents. In the following, the usage of the recovery features in ERIS and the resulting PRISM code generation for these is explained. Therefore, the system introduced in Section 6.4, Figure 6.2 is used as working example.

As observed in the node settings window (Section 7.1.4), the user may choose via checkboxes whether the concerned node is capable of recovering from a defect or a corruption. Based on this choice, the window elapses and gives the possibility to declare a success rate, identically to the failure and intrusion rates as before. Given that, ERIS generates further constants to be used in the transitions. Below

Listing 7.11 shows this exemplarily for the recoverable node n_3 of the previously introduced system.

```
12  const double rn3DEFREC = 0.05;
13  const double rn3CORREC = 0.05;
```

Listing 7.11: Recovery Constants

In addition to this success rate, a recovery strategy must be chosen. The available strategies cover the previously determined concept: `General` corresponds to the *self-performed* strategy where the component has implemented some mechanisms that lets it recover by itself to the provided rate. Listing 7.12 shows such a general recovery transition from the `corr` state to the `ok` state for node n_3.

```
42  [] (n3=2) & (operational) -> rn3CORREC : (n3'=0);
```

Listing 7.12: General Recovery from Corrupted State

This reads: *Whenever node n_3 is in the* `corr` *state and the system is still operational, there is a transition indicated by rate rn3CORREC that node n_3 turns back into the* `ok` *state.* The complete code for this strategy can be found in Annex A Listing A.12.

`Restricted` accords to the strategy *performed by the outside*. Exactly as determined earlier, the recovery of the node may only occur to the given probability rate in the case that a non-disrupted path to the environment node still exists. To accomplish this, ERIS recursively searches for all paths from the selected node back to the `Env`-node. These paths are stored in a dedicated formula as it can be seen in Listing 7.13.

```
23  formula pathesn3DEFREC = n1=0 & n2=0;
24  formula pathesn3CORREC = n1=0 & n2=0;
```

Listing 7.13: Generated Restricted Recovery Paths

Since the only path for n_3 to `Env` is via n_1 to n_2, these two nodes were collected and are required to be in the `ok` state. Additional paths would be connected via the logical or (|) operator. Note that in this example, n_3 can recover from defects and corruptions with the restricted strategy. Now to apply this strategy the path formula is used as a precondition for this transition to be taken, as shown in Listing 7.14 for the defect recovery.

```
46  [] (n3=1) & (operational) & (pathesn3DEFREC) -> rn3DEFREC : (n3'=0);
```

Listing 7.14: Restricted Recovery Transition from Defective State

The code for this strategy can be found in Annex A, Listing A.13.

And lastly, the `Custom` option can be used for the *performed by another component* strategy. The implementation of this strategy is a bit more powerful than its theory: Selecting this strategy opens a small input field. Similar to the node dependencies, the user can define a condition under that the concerned node recovers to the provided probability rate. For instance, the node may only recover if some other nodes are still functional. That being so, the user can actually determine multiple nodes to be operating correctly, rather than having one node n_{rec} as before. However, the responsibility of specifying undisturbed Reach-paths is thereby transferred to the user: Unlike the restricted strategy, ERIS does not collect all existing paths to the defined nodes and checks whether they are intact. This allows us a more flexible modelling of the recovery strategy, yet also permits the specification of recovery nodes without undisturbed Reach-paths to the affected node. Thus this strategy has to be handled with care in practice. Listing 7.15 shows this for the recovery performed by the watchdog node n_5.

```
24  formula pathesn3CORREC = n5=0;
    ...
49  [] (n3=2) & (operational) & (pathesn3CORREC) -> rn3CORREC : (n3'=0);
```

Listing 7.15: Custom Recovery Transition from Corrupted State

A path formula is defined that blindly transfers the user input. Hence, the user input is checked to be syntactically correct, however, the path semantics are deliberately left untouched to allow the described flexibility. In Annex A, Listing A.14 the complete code for this strategy can be found.

7.2.4 Modularization

The dependency graph modularization is implemented in ERIS in the following way. In the node settings window (see Section 7.1.4) of a selected node, the user can define whether this node is actually representing a module. Therefore, two options are available: A simulation model or an ERIS dependency graph module. The simulation model leads to the hybrid evaluation scheme previously defined in Section 5.2.3 and the ERIS model leads to the recursive analytical approach of Section 5.2.2. Its implementation in ERIS is presented in the next Section 7.3. Note that since for any module evaluation it is essential to know the time points that are meant to be evaluated in the original system, only the experiment evaluation can be used for modularized systems. Consequently, the normal transformation process will disregard from any module nodes and solely view the focused system.

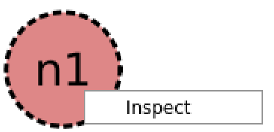


Figure 7.6: ERIS Module Node

In case an ERIS module is selected, the program expects a well-defined ERIS dependency graph in terms of an XML file that, for instance, has previously been modelled in the GUI. Once a valid file has been supplied and the node settings are saved, the node changes its stroke style into a dashed line to indicate that it is representing a module. Now the user can view the module by right-clicking on the node and proceeding the `Inspect` action as shown in Figure 7.6. This action opens a new tab next to the main modelling tab and loads the provided dependency graph model. This tab works like an independent Model tab, meaning that the user can adjust, modify and store the model, as well as trigger an independent evaluation process for it. At this moment, it is not possible to open a new Model tab to create dependency graphs in parallel. In case of a simulation model, ERIS expects an independent simulation model to be provided in terms of the GNU Octave[2] language or Octave-compatible MATLAB. Similarly to before, the node will change its stroke style, this time to a dash-dot line, indicating it is holding a simulation model. However, this module cannot be inspected, because ERIS is only supporting the modelling of dependency graphs and so far a generation of dependency graph-based simulation models has not been developed. As a result the user must make sure that the provided program is correct beforehand. During the evaluation process, the simulation module is evaluated using Octave and ERIS expects that after the process has ended successfully, the results will be available on the standard output.

7.3 Evaluation

The performed evaluation process depends on the modelled system and the chosen tool bar option (see Section 7.1.2). Since the transform button only generates a PRISM based Markov model and the play button opens the PRISM GUI, where the user must perform the evaluation with the settings available in PRISM, subsequently it is focussed on the experiment evaluation that uses the model checker and/or simulation tool in the background.

After the PRISM module definition, we could see a declaration of three `labels`, as displayed again in Listing 7.16, that were automatically generated.

[2] https://octave.org/ visited on 17th July 2023

```
46 label "systemfailure" = !operational;
47 label "defective" = (n1=1 | (n2=1)) | (n3=1) ;
48 label "corrupted" = n1=2 | n3=2 ;
```

Listing 7.16: Generated Labels

The first label `systemfailure` accords our basic property of reaching the Fail-state by negating the formula generated with respect to the mode of operation. The labels `defective` and `corrupted` concern the safety- and respective security-wise failure of the system. While we are generally not interested in separating the failure kinds, it is necessary for the analytical module evaluation (see Section 5.2.2). In the evaluation process these labels are used within the CSL formula yielding something like $P^{=?}$ $[F^{=T}$ "$systemfailure''$] (with T being some point in time). PRISM uses this formula to check the Markov model for some provided $t \in T$, computing the probability.

The experiment process scheme on a modularized system is shown in the sequence diagram of Figure 7.7. Firstly, the user must provide information on the evaluation process via the `Evaluation Options` in the `Options` menu (see 7.1.1). These comprise

- the evaluation property,
- the interval,
- and the steps.

For instance, the `systemfailure` property in an interval of 12 time units and steps of 1. During the dependency graph transformation ERIS searches for any module nodes. In case a module node is found, the module evaluation is triggered before proceeding with the original graph. Naturally the module evaluation between ERIS and simulation models differ:

ERIS Model If the found module is an ERIS model, the user-specified interval and steps are used to evaluate the module in terms of a PRISM experiment. Therefore, ERIS generates a .pctl file containing the selected properties and triggers the experiment process via the command line. Once all selected time points have been evaluated, ERIS collects the results from PRISM, reinterprets them as rates and stores them internally in association to the concerned module node.

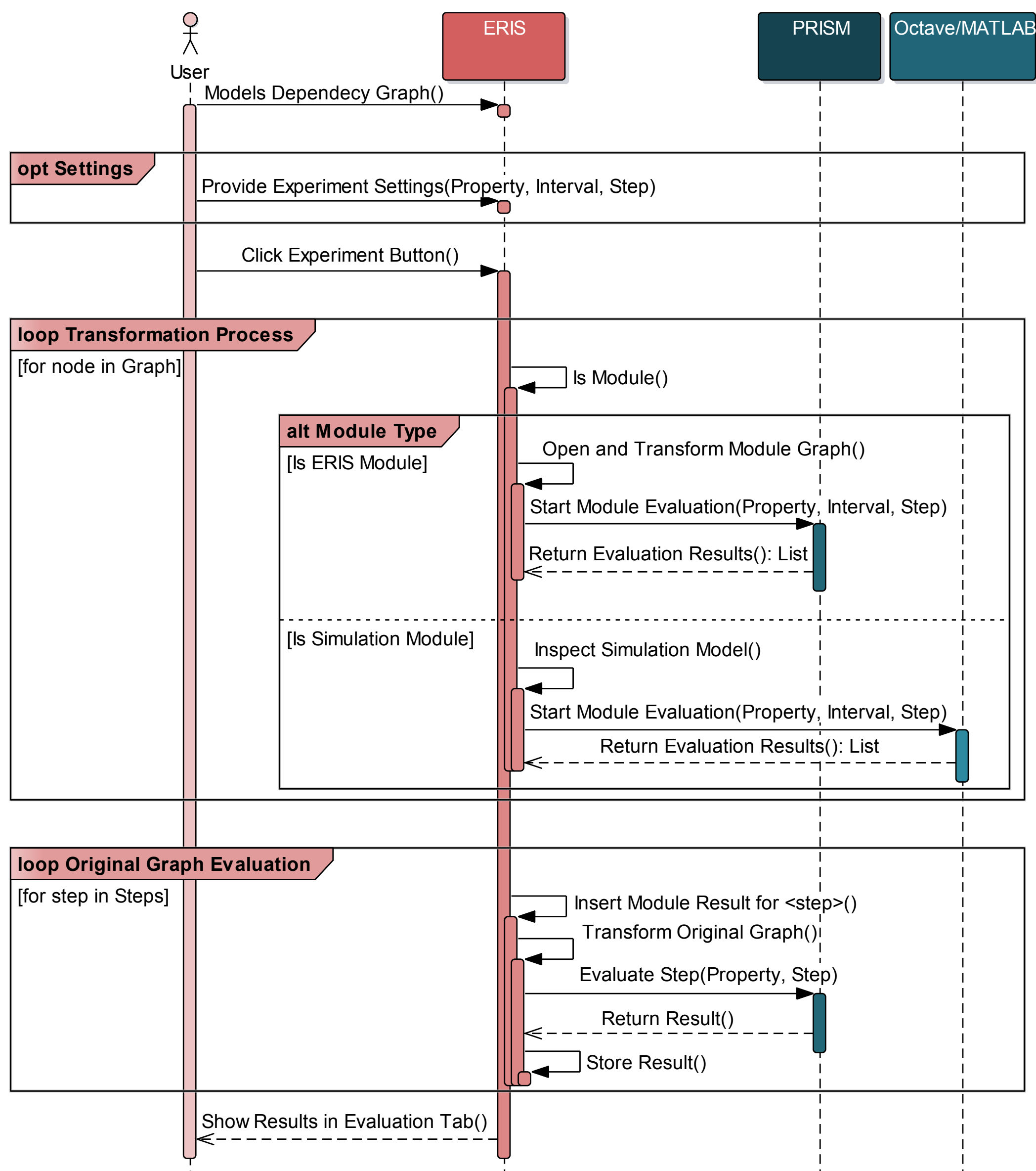


Figure 7.7: Module Evaluation Process

Simulation Model (Hybrid) If the module is a simulation model, the hybrid evaluation is performed. Therefore, ERIS was previously combined with a simulation-based tool called *AT-CARS* [RH20a; RHK21]. AT-CARS uses a different kind of modelling, where the system's reliability structure is inherited as a Reliability Block Diagram (RBD) and is transferred into a state-based calculation model. This model is evaluated by making use of Monte Carlo Simulation (MCS). A detailed description of this approach can be found in the next chapter, Section

8.2. Following the MCS principle (see also Section 2.5.2), the system behaviour is simulated and the point in time and the belonging state of each transition recorded. The recorded results are used to analyse the desired safety KPI, in our case, the failure and attack rates. Since the MCS is performed with MATLAB or Octave, ERIS expects an Octave compatible simulation model (to maintain licence independence) deposited via the node settings. Then, Octave is triggered in the background via the command-line with that model. In the current implementation ERIS expects the simulation model to contain the experiment information. Unlike the analytical model, the simulation process is only performed once, to optimize its performance and in accordance with the already detailed MCS process. Its concluding analysis is configured to yield the rates for the desired time steps. ERIS waits for the process to finish and then collects the output results, expecting a list of tuples that associate time point and rate, equivalent to the stored result of the analytical module.

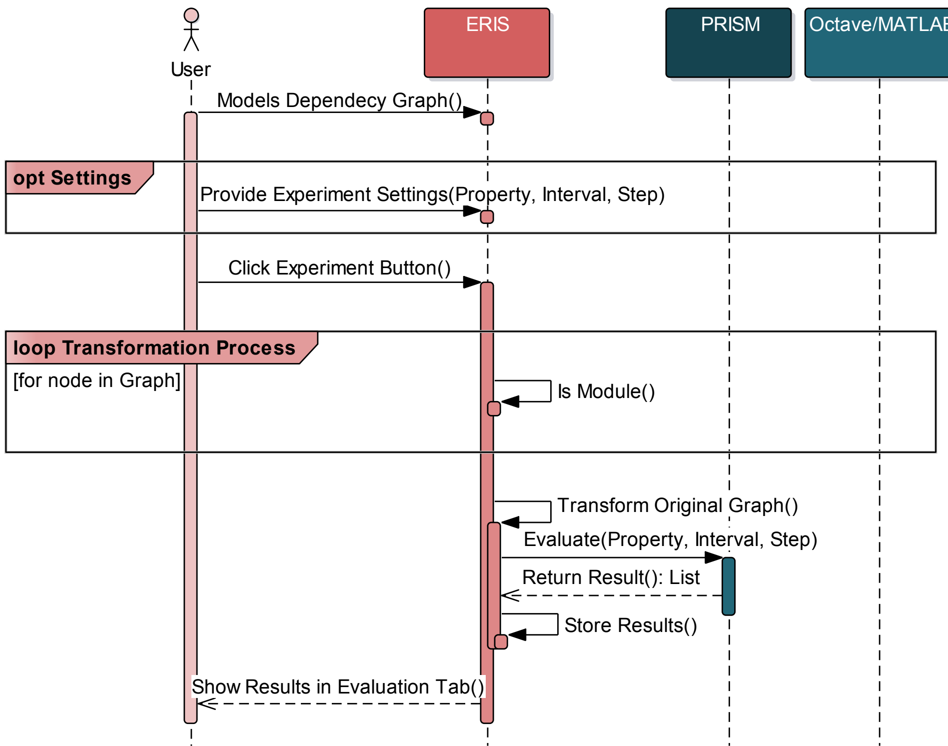


Figure 7.8: Evaluation Process of a Non-modularized System

This procedure is performed until no more modules are found. Afterwards, the PRISM experiment process is triggered for the original graph. Since we have to consider the inputs generated by the module evaluation in each evaluation step in the original graph (see also Chapter 5 for the reasons), the experiment pro-

cess must be performed step-wise. Thus, for each step in the defined interval, the node's failure and attack rates are reset according to the obtained results and the Markov model is rebuilt. Then the experiment process is triggered for one step and the PRISM results obtained. Once the end of the interval is reached the collected results are plotted in the evaluation tab of the ERIS GUI. Via right-click the user may export the results in the CSV or PDF format. At the current state only depth 1 modularizations are supported. Thus, modules that contain module nodes cannot be evaluated automatically with ERIS. In case the experiment evaluation is triggered for a non-modularized system, the process is much simpler, as shown in the sequence diagram of Figure 7.8. With the absence of modules, it is not necessary to transform the graph step-wise and instead the entire experiment evaluation can be performed in one PRISM call in the background. Note that since the original graph cannot be a simulation model, the MATLAB/octave evaluation becomes obsolete.

7.3.1 Performance

While the transformation process from a dependency graph into a Markov model is a very quick task, the evaluation of modularized systems is in general a bit slower than non-modularized systems (considering equal input systems and evaluation properties). This owes to the fact that the process scheme of modularized systems affords more transformations and single-step evaluations, which require individual program calls and the model to be constructed for every step, even though its architecture did not change. Since the evaluation task is performed by PRISM or Octave/MATLAB, the evaluation performance depends on these tools and the hardware they run on. PRISM has implemented different engines which can be manually selected. It is stated that typical computers can handle models with a state size of 10^7 to 10^8 making use of the *hybrid* engine[3]. The evaluation time for smaller models is very fast (a few milliseconds to seconds) but increases heavily with a growing model size. An example for this can be seen in the next Section 7.4. Regarding simulation, as discussed before, it is extremely difficult to determine the time that is needed to achieve satisfying results. Thus, the evaluation time is also very model-dependent without being connected to the model size. However, in almost all cases it will take longer than model checking, which must be taken into account when considering whether or not to use the hybrid.

[3] See https://www.prismmodelchecker.org/manual/ConfiguringPRISM/ComputationEngines visited on 29th October 2023

7.4 Example Analysis

With tool support and the power to handle the exponential state growth through modularizations we are now prepared to analyse realistic systems. Therefore, we go back to the earlier introduced system of an autonomous vehicle, described as a dependency graph in Section 4.1.1. We saw that this system leads to a massive state generation in the Markov model. Precisely, the ERIS generated PRISM code leads to 1 792 068 717 312 states and 4 819 816 742 400 transitions in the CTMC. Thus, performing an evaluation with regular computers easily results in memory issues, or an extremely high evaluation time. Consequently, this system profits from modularization.

Multiple options for splitting this system are conceivable. Figure 7.9 visualizes three convenient possibilities. All three options represent modules that are rather encapsulated subsystems. However, in the earlier studies of well-defined modularizations of Section 5.3, we learned that lengthy Reach-paths pointing to the module should be avoided. Now both CAN bus subsystems have the problem of containing a rather long Reach-path from the Env-node. The comfort CAN bus (option 1) models less components, however, consists of more uncritical nodes than the motor CAN bus (option 2). As a consequence, its state space is larger and thus a modularization would lead to a higher decrease of Markov states in the abstracted graph. The SDS module (option 3) has the same Reach-path (via the Gateway-node), but it also contains an additional, shorter Reach-path via the VANET Router. It can be argued that as long as this Reach-path is available, it resembles an instant reachability closer and thus leads to less over-approximation. However, this effect is presumably rather small and depends on the defined attack rates. Moreover, this option would decrease the state space most effectively, as it contains the highest number of nodes with a large amount of uncritical ones.

The functional relationships between the SDS's computing nodes and the various LIDAR, RADAR and camera sensors permits the modularization (a split of the mode of operation is possible). However, the functional relations must be defined so that the GPS node n_7 can be excluded from the module, because it also has a functional dependency to the navigational system (node n_2). Otherwise, it would be required to occur in both graphs and a modularization would not be possible. This is very reasonable as we can argue that none of the LIDAR, RADAR or camera sensors are capable of delivering an output equal to the GPS sensor. Yet, without any redundancy, this makes the GPS sensor a single point of failure. Table 7.1 gives an overview on the performance indicators of the different modularization options when built with ERIS. Any of these options are viable in theory, however, option 1 and option 2 still lead to a massive state generation in the abstracted graph, making

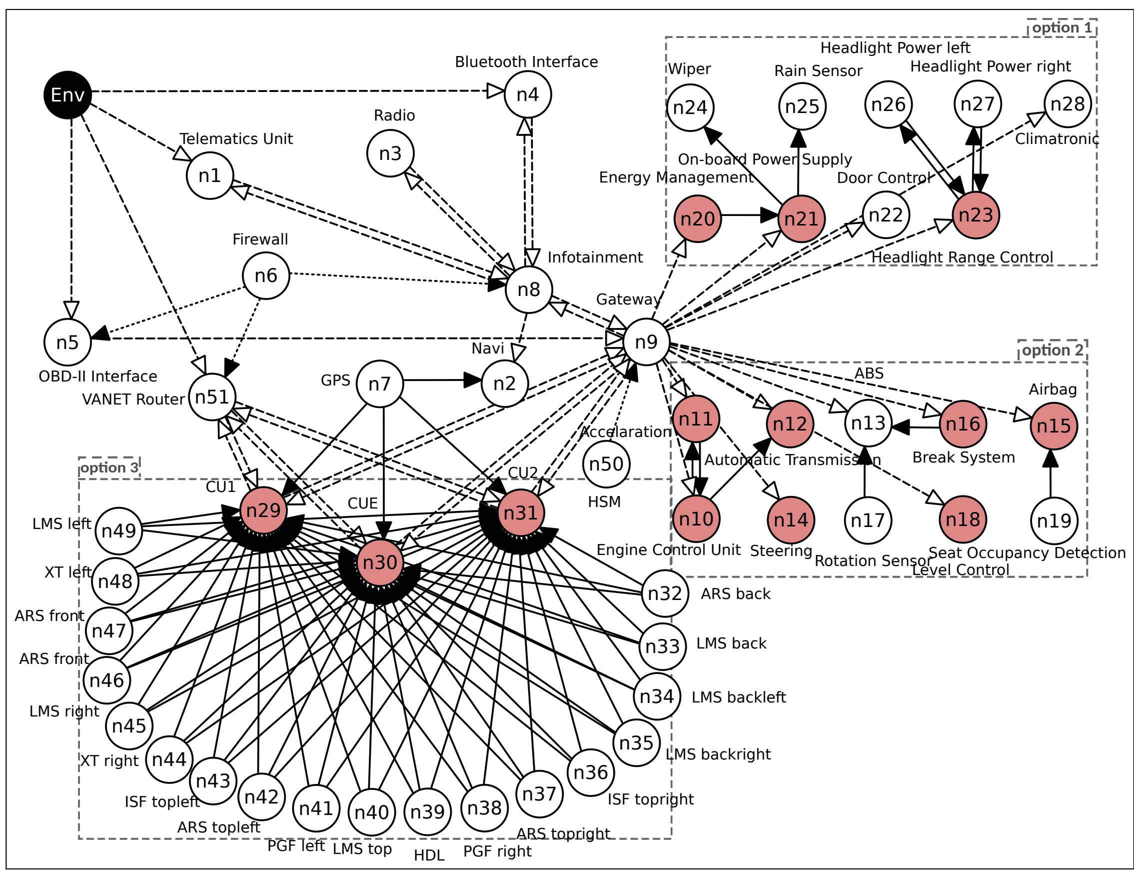


Figure 7.9: Modularization Options of Autonomous Vehicle Dependency Graph

the evaluation with regular computers extremely difficult. Option 3, however, lies with 10^8 Markov states in the range of computable systems, as discussed in the previous Section 7.3.1, and thus has been chosen.

ERIS can easily model the sensor redundancy. Though, regarding the computing units, only a flat redundancy definition is possible: While in practice the emergency computing unit CUE acts as some sort of fallback system that is independent from other CU but activated once they fail, in ERIS we could only define all three nodes as equally redundant. In the same thought, if we were to model the software applications running on these computing units they could only be represented in a static manner, belonging to the CU as defined by functional dependencies. However, in reality a switching between these software instances can be possible, as it was presented in earlier sections of this thesis. Owing to this and the fact that the simulation tool AT-CARS is actually able to model these details (see also Section 8.2), we make use of the hybrid evaluation approach.

Table 7.1: Model Sizes of the Different Modularization Options

	States		Transitions	
	Abstracted	Module	Abstracted	Module
Original	1 792 068 717 312		4 819 816 742 400	
Option 1	1 783 783 296	324	4 480 103 520	684
Option 2	8 287 629 696	96	22 046 997 312	189
Option 3	87 913 152	149 688	221 818 496	401 436

In order to analyse the system, dummy failure and attack rates have been established as listed in Table 7.2. In addition to that, some recovery rates and also

Table 7.2: Assumed Failure and Attack Rates

Component Type	Failure Rate $\frac{1}{h}$	Attack Rate $\frac{1}{h}$
CAN components	$1.3 \cdot 10^{-7}$ (critical) $1.2 \cdot 10^{-6}$ (uncritical)	$1.2 \cdot 10^{-5}$
$CU1$ and $CU2$	$1.02 \cdot 10^{-6}$	$1.1 \cdot 10^{-6}$
CUE	$1.02 \cdot 10^{-7}$	$1.1 \cdot 10^{-7}$
AI Sensors	$1.01 \cdot 10^{-6} - 1.95 \cdot 10^{-7}$	
Other	$1.1 \cdot 10^{-5} - 1.1 \cdot 10^{-7}$	$1.5 \cdot 10^{-2} - 1.9 \cdot 10^{-6}$

security guarantees have been defined. The precise definition can be withdrawn from the PRISM code Listing displayed in Annex A.3. As a demonstration our usual property, the system failure probability, is evaluated for 24 months of continuous operation. Therefore, the module simulation was performed in cooperation with the authors of AT-CARS. Thereby, 100 000 iterations where run without parallelization which took around 336 minutes. As a demonstration of the simulation procedure, one of these iterations is exemplarily shown in Annex B.3, Table B.6. Due to the large state space, the subsequent model checking of the abstracted graph was also rather timely with approximately 229 minutes. Note that the computations were performed on a regular computer[4]. Despite not being comparable by employing a more abstract model, it must be noted that an analytical evaluation of the module with ERIS, to that shown specification, only takes approximately 4 seconds for all 24 steps.

Figure 7.10 visualizes the results by plotting the system failure probability on the left hand side and the respective reliability/availability drawn from it on the right hand side.

[4]Lenovo T14 intel i7-118G7 (3 GHz) and 24GB RAM on WSL Debian 11

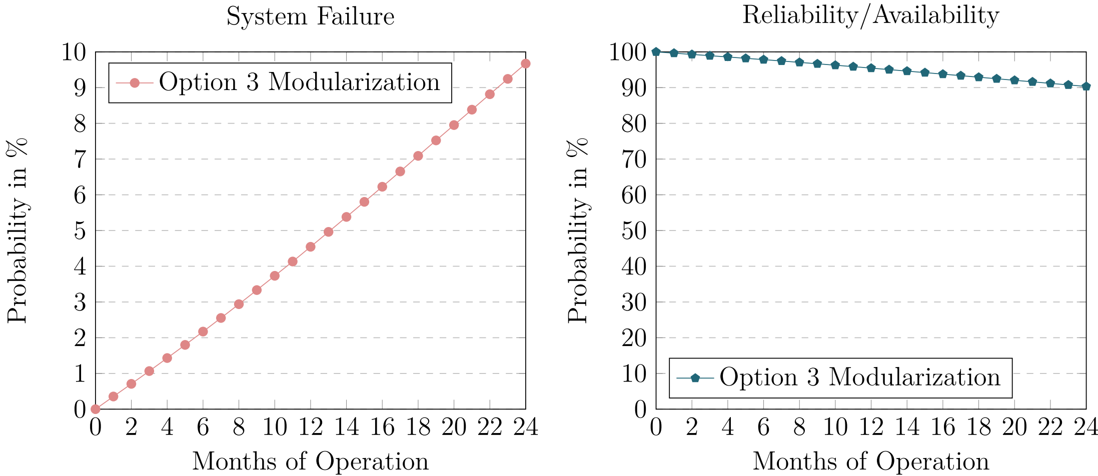


Figure 7.10: Plot: Option 3 Evaluation Results with AT-CARS Module

After 2 years of operation the probability that our system is still operational lies at around 90%. Precise results can be found in Annex B.3 with Table B.4 depicting the module rates computed by AT-CARS and Table B.5 the computed failure probability by ERIS.

Now apart from the usually discussed availability and reliability assessment of the system, our method could also be used as basis to performing a sensitivity analysis. Therewith, for instance, critical attack paths or components that are particularly endangered could be identified. As an example, we try to identify the most probable attack path to the gateway and the SDS. Several paths to reach these components exist: Via the VANET router, the OBD-II interface, or the infotainment system, on behalf of previous corruptions of either the telematics unit or the Bluetooth interface. In order to measure their vulnerability, we compute the probability that either of these three components turns corrupted by evaluating the following CSL properties: $P^{=?}_{OBD}\ [F^{=T}\ n_5 = 2]$, $P^{=?}_{Infotainment}\ [F^{=T}\ n_8 = 2]$ and $P^{=?}_{VANET}\ [F^{=T}\ n_{51} = 2]$ again for 24 months of operation. Naturally a proper sensitivity analysis would be subject to an intensive study of possible events and the thereby following determination of appropriate evaluation properties. This attempt is merely a demonstration of this method's application capabilities. In many cases the prior knowledge of the specified attack and failure rates makes the result tendency of such an analysis predictable. However, the analysis views the system as a whole and especially if varying Reach-paths and different nodes that supply security guarantees, failing to individual rates, are modelled, this kind of analysis can be very supportive.

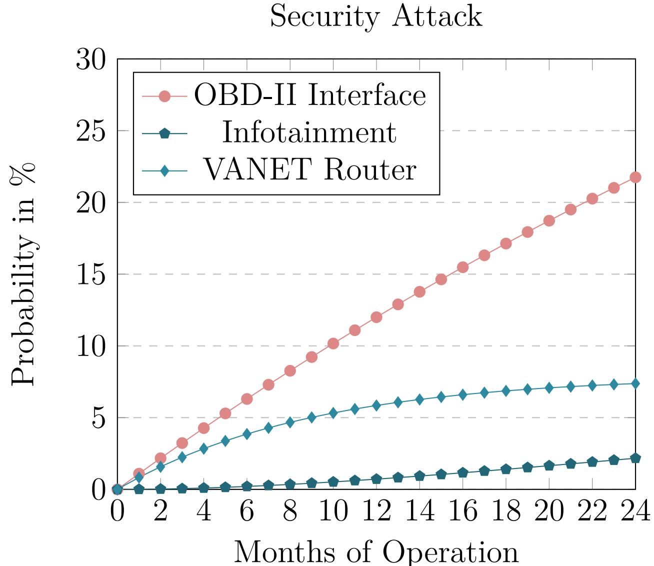


Figure 7.11: Plot: Option 3 Corruption Probabilities for the Sensitivity Analysis

Figure 7.11 plots all three security attack probabilities, based on the computed results that can be found in Annex B.3. We can see that the probability that the OBD-II interface turns corrupted is by far the highest, making it assumably the most probable attack path. This is explained by the fact that its attack rate was set the highest, and in comparison to the infotainment system, it is directly reachable and therewith its corruption does presuppose a prior corruption of other nodes. Since all three nodes receive the same security guarantees by the firewall, it plays a negligible factor in the comparison. Of course such a high attack rate is quite unrealistic, however, as mentioned before, the OBD-II interface is often not well protected. If the physical protection ceases to apply in addition to the already sparse cryptographic protection, perhaps if we view autonomous taxis rather than customer vehicles, the vulnerability of this component increases drastically. Apart from that, we can see that the attack probability of the VANET router seems to level off at $\approx 7.5\%$. This is attributed to its ability of recovering from a corrupted state.

8 Related Work

The challenge of assessing the safety and security of autonomous vehicles is in constant rise due to the rapid advancements of technology. In comparison to classical vehicles, new parameters and failure sources must be taken into account. Therewith, some of the formerly applied techniques have become obsolete. As a consequence, new assessment approaches, tailored to these systems, have been developed and traditional methods have been reworked. This chapter introduces and compares approaches related to the work of this thesis.

In 2015 Kriaa et al. [Kri+15] gave an extensive overview on approaches that integrate or unify safety and security in different product lifecycles. Even though this review is not targeted on the automotive domain but on industrial control systems, it proposes an industry independent categorization and consideration of safety and security. This categorization mainly separates between (i) generic approaches and (ii) model-based approaches. Generic approaches regard safety and security on a very macroscopic level of system design and risk evaluation and thus consider, among others, the unification and the alignment of standards and processes. For example, work like [Cui+19] that presents a framework for the collaborative safety and security analysis of autonomous vehicles. This kind of alignment of standards and processes builds a significant basis for the justification of any safety and security models, however, due to their high-level perspective they are incomparable to this work and are not further detailed here. Model-based approaches, on the other hand, are specified by relying on a formal or semi-formal system representation [Kri+15] which accords the methodology of this thesis. Various developments fall into that category, for instance, models that are built upon the combination of the safety-oriented FTA with the security oriented Attack Trees such as presented by Kumar and Stoelinga [KS17]. Other approaches are directly focussed on the interdependencies of safety and security, such as work based on Boolean logic Driven Markov Processess (BDMPs), a combination of fault trees and Markov Processes, by Piètre-Cambacédès and Bouissou [PB10]. In order to relate to the methodology of this thesis, the viewed approach must be capable of modelling the system in a similar manner with a similar evaluation goal. Therefore, the subsequent sections focus on two methods that satisfy this condition.

8.1 Markov-based Approach

The authors of [Här+23] propose a Markov-based approach for the safety and reliability analysis of autonomous vehicles. Similar to our modelling, the Markov states represent the health states of the modelled system components and a transition relates to a single component's failure or repair. At the current stage of this research security is not concerned, though the modelling would generally allow for its extension. Consequently, their Markov model is described by a state space of 2^K with K being the number of modelled components and 2 referring to the allowed states *operational* and *failed* (see also [Här+23; Wal+23]). This is equivalent to our modelling in cases where only safety is considered. However, the authors mention a refinement that introduces an additional degraded state which would increase the state-space accordingly to 3^K, if we consider these states for every modelled component. In the referred publication, the method is demonstrated by modelling a sensory system of an autonomous vehicle. Despite being capable of considering redundancies, the multiplicity of sensors is not covered in this example. The main difference to our methodology is that the modelling takes place on behalf of the Markov chain. This means, there is no formal specification of a superordinate modelling layer, like the dependency graph, where the definition of the system's Markov chain is drawn from. Thus, all system specifications, such as redundancy definitions, functional relations and the system's operational capabilities, must be considered during the modelling process of the Markov chain. This can make the modelling process an error-prone and tedious task, especially when it comes to the analysis of large systems, or an analysis with varying architectures. Furthermore, while attacks could potentially be modelled in the manner of failures (occurring at a defined rate or probability), the method would require an extension to include possible entry points as well as attack paths a potential attacker could exploit, if security should be considered.

The behaviour of component failures and repairs can be modelled by two different kind of rates: Exponentially distributed rates, like in our method, and time-dependent rates, where the rate can change during the lifetime of the analysed system or part. This can be beneficial to reflect ageing effects or the improvement of functions, e.g., the growing maturity level of software through updates, and goes beyond the capabilities of our method. The model evaluation is performed by making use of a dedicated Python library. Thereby, it is mentioned that 10 000 Markov states are somewhat a limit, when an evaluation time under an hour is envisaged [Här+23]. Given this statement, it seems that the evaluation method is much slower than the implementation (with the PRISM backend) in ERIS: Reviewing the example analysis of the previous chapter, the module evaluation with 149 688 states only required 4 seconds. Though it must be mentioned, that the

evaluation time heavily depends on the regarded properties, input rates and used hardware, which makes it difficult to compare the approaches performance-wise. Despite that, by not transferring the evaluation responsibility to a model checker, the authors are able to identify critical components more easily by viewing the entry of states related to their failure. With ERIS this is not as simple, because the responsibility of creating the state space is handed over to PRISM, which makes the repatriation from single states to nodes currently not possible. In further research [Wal+23], the authors propose the same solution to encounter the exponentially growing state space: Modularizations. Even though the approach does not make use of an superordinate modelling layer, the subsystems to form the modules are performed with respect to the functional relations of the system. Therefore, the modularization yields three systems: “Complex system”, “Module” and “Modularized system” which correlate to our specification of (the Markov model’s) “original graph” “module graph” and “abstracted graph”. In contrary to our method, these modules can only be evaluated by Markov analysis and a hybridization or the combination with other methods and techniques has not been considered.

Concluding it can be said that very strong parallels to this approach exist. However, while in some directions it can be considered more mature, for instance, the consideration of time-dependent failure rates or the direct connection from Markov states to the modelled components, others, like the modelling of security implications and attacks, are being neglected entirely. Ultimately, the lack of a superordinate modelling, like the dependency graph, is limiting the modelling possibilities and it is questionable whether complex tasks like large system modularizations can be performed correctly and reproducible in this way.

8.2 AT-CARS

In [Hei+19] a state-based Monte Carlo Simulation (MCS) approach for the safety analysis of hard- and software components of autonomous vehicles has been presented. In later publications, the software tool resulting off of this approach is referred to as *AT-CARS* [RHK21]. The main goal of this approach is to support the analysis of autonomous vehicles by an improved focus on its fail-operational behaviour. As presented earlier in this thesis, the key for achieving a fail-operational behaviour is a suitable redundancy concept combined with a failure management process. Thereby, the concept presented in Chapter 2.2, where computational resources in terms of software applications are being switched between multiple (redundant) computing units, are on the rise. With AT-CARS, the authors contribute to the

safety verification of autonomous systems by developing a modelling technique that takes these dynamic operations, which are actually dynamic changes of the architecture, into account.

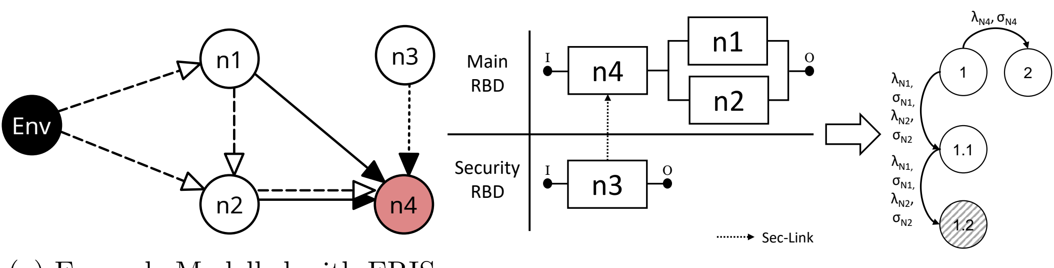
To achieve this two kinds of models are used: The reliability structure of the concerned system is captured by a RBD-oriented modelling which is transformed into state diagrams for the simulation process. This state-based model is oriented on a Markov model but adjusted to avoid state-space explosions. Since AT-CARS does not offer GUI support, the user must provide the input system specification in terms of multiple JavaScript Object Notation (JSON) files. The RBD-oriented structure is an abstract system description, like the dependency graph, and the state diagram acts as the analysis basis, like the Markov model of this thesis. In contrary to our method, a lower level of abstraction is envisaged: Hardware components and software elements are explicitly modelled in separate: Hardware components have a defined failure behaviour and are equipped with one or more computing nodes, that provide computational resources. Software elements are specified with computational resources to *consume* and with a failure behaviour of their own. These software elements cannot run on their own and must be assigned to a hardware element. The idea is, that a software element can only run on a hardware element, as long as the required computational resources can be provided to it. This is key to modelling the addressed dynamic behaviour: In case a hardware component fails, its computational resources extinguish and the software applications previously placed on it cannot run any more. However, in case other active hardware components with computational resources are available, the software application can be switched to run on these (if specified). In that way, system level failure management processes like FDIR can be reflected. While it is possible to model software and hardware as individual nodes in ERIS and also consider a functional relation between them, it is not possible to dynamically change these relations as it would be the case if a software application was switched to another hardware.

The failure behaviour of hardware and software components is described in terms of probability distributions similar to our method. However, in addition, AT-CARS is designed to allow the definition of multiple failure modes for one component. While this is possible in our Markov model as well, the input interface of ERIS does currently not provide for it, apart from the distinction of safety and security failures. With that the definition of different kinds of failure models such as exponentially distributed failures models, Weibull distributed failure models etc. becomes possible. In order to consider the entire FDIR process in AT-CARS, also component recovery has been implemented, which is, however, only considered for software applications, because the authors assume that hardware devices cannot

recover [Hei+19]. Since in ERIS the distinction between software and hardware must be made by the user, hardware recovery, which is possible in reality as we saw in Section 6.1.1, can be modelled.

By making use of simulation as the evaluation method, AT-CARS follows a different paradigm (see also Chapter 2.5.2). As a result, the approach suffers the usual problems of stochastic simulation: The evaluation time required to receive a good result estimate cannot be determined beforehand and the result accuracy depends on the number of performed iterations. Consequently, the simulation time can be very high to achieve accurate results. However, the simulation time and the model size usually do not correlate. Therewith, other than our Markov analysis, also very large models can be evaluated without encountering problems due to an unmanageable state space. In the previous chapter it was shown how this benefit and the fact that AT-CARS employs a more fine-grained modelling can be used to create a hybrid model of ERIS and AT-CARS. The foundation for that was laid in [RH20a].

The first steps to do so were to make sure that both approaches, regardless of using a different modelling and evaluation method, are in fact calculating the same or rather similar results for the same inputs to avoid information loss with the hybridization. Therefore it was necessary to introduce the modelling of security attacks into AT-CARS. Since in the current modelling there is no option for defining the implications induced by Reach-dependencies, these implications have to be modelled manually for the considered system. Then, security attacks are simply modelled as an additional failure mode. Now to survey whether both approaches are working in a similar nature, the example system shown in Figure 8.1 has been evaluated. Figure 8.1a shows the ERIS model of this system and Figure 8.1b a



(a) Example Modelled with ERIS

(b) Example Modelled with AT-CARS

Figure 8.1: Small Example for Comparison

visualization of the according AT-CARS model.

As before, critical nodes are highlighted in red. The two nodes n_1 and n_2 represent non-critical, data supplying components which may be reached from the outside.

Additionally, node n_1 can reach n_2, and node n_4 can be reached from node n_2. Node n_3 cannot be reached at all but provides security guarantees to the node n_4. We defined nodes n_1 and n_2 to produce redundant data to n_4. This yields the following mode of operation: $\phi = \hat{n}_4 \wedge (\hat{n}_2 \vee \hat{n}_1)$.

Figure 8.1b demonstrates the reliability structure of the system modelled in AT-CARS. The parallel structure of the RBD-oriented modelling represents that at least one of the components modelled by n_1 or n_2 must be operational to satisfy the mode of operation. The respective node n_4 is modelled in series to the parallel structure of n_1 and n_2 to indicate that a failure of n_4 leads to a system failure. To model the security guarantees provided by n_3, the attack rate on n_4 is defined to change accordingly if n_3 fails. The resulting state diagram consists of four states with the following definitions:

- State 1: n_1, n_2, n_4 are functional
- State 1.1: n_4 functional, n_1 or n_2 failed
- State 1.2: n_4 functional, n_1 and n_2 failed
- State 2: n_4 failed

Dummy failure and attack rates (exponentially distributed) were assumed. Precise values and the generated PRISM code can be found in Annex A.4.

Table 8.1: Results of the Comparison

Hours	ERIS States: 30 Transitions: 65 Time: 23ms	Simulation States: 4 Transitions: ∅3.31 Time: 8764s	Δ
0.00	0.00%	0.00%	0.00%
1.00	19.26%	19.37%	0.10%
2.00	40.05%	40.44%	0.39%
3.00	57.51%	57.73%	0.22%
4.00	70.72%	71.02%	0.30%
5.00	80.18%	80.49%	0.31%
6.00	86.74%	87.18%	0.44%
7.00	91.19%	91.46%	0.27%
8.00	94.18%	94.44%	0.26%
9.00	96.17%	96.32%	0.15%
10.0	97.48%	97.59%	0.11%

For the comparison, the probability that the system enters the fail state at a given time point (specific hour) was evaluated in either model. The simulation has been performed with 100 000 iterations. The computed results are listed in Table 8.1. We can observe that even though the results are not identical, the anomaly Δ is never beyond 0.44%. Concluding it can be said that the simulation delivers a good estimation of the analytical solution of ERIS and thus these approaches can be combined without encountering unreasonable information loss.

9 Conclusion

The safety and security analysis of autonomous vehicles is paramount. It involves a comprehensive evaluation of functional safety, cybersecurity, the compliance with standards, sophisticated testing procedures and permanent progress monitoring. In consequence, a multitude of methods and tools are needed to yield a holistic assessment approach. This dissertation presented one of them.

Subsequently, a thesis summary is given and the application possibilities of the methodology are highlighted once more. Afterwards, the main challenges of this research are addressed. At the end, the application of this approach beyond the automotive domain is discussed shortly and a view on future work is given.

9.1 Summary

The developed methodology was inspired by the new architectures of autonomous and automated vehicles as well as the rising challenges to verify their safety and security. This verification is key to gain the approval of legislators and authorities, but also to establish trust of autonomous driving in the general public. The superordinate goal was to close a gap between widely adopted approaches that target either safety or security and instead establish a methodology that can handle both properties on an equal level and further, consider their intertwined effects. Therefore, a graph-based model was developed that views safety and security relations between components and distinguishes between attacks and failures of those. Given that critical systems must be able to act fail-safe and especially since autonomous systems are thought to be able to become self-aware, the model was extended by the consideration of different kinds of repair and recovery actions. In this way, both the self-induced recovery of components and the recovery triggered from the outside could be included. Therewith, a playground to reason about the interdependency of single components during failures and attacks, by ultimately viewing their consequences to the system's functionality, was established. This is essential for enabling a holistic system assessment mandatory to capture the system's failure behaviour realistically.

To perform this assessment over a specified period of time, such as the envisaged lifetime of the system, a transformation from the static graph into a quantitative model in terms of a CTMC was specified, making a quantitative analysis via, e.g., probabilistic model checking possible. While Markov models proved to allow an accurate and fast evaluation, they also introduced an exponentially growing state space in relation to the captured components. With the goal of being capable of analysing vehicular system structures, it became clear that this exponential state growth could easily become the bottleneck of the methodology, if unhandled. To conquer this problem, a modularization scheme for dependency graphs was developed and it was demonstrated that it enables the analysis of very large systems. Finally, the implementation of this methodology into a tool named ERIS was presented, providing for a comfortable application.

Despite the main objective of analysing the system failure probability it is possible to analyse further system or subsystem properties that are expressible in PCTL or CSL. Therewith, weak points in the architecture can be narrowed down, if not apparent due to the provided input rates, as it was demonstrated in the example analysis. In regard to the approval of an autonomous vehicle, the reliance on these input failure and attack rates being realistic in order to obtain legitimate results is perhaps the biggest downside of this approach. As discussed earlier, determining the failure behaviour and deriving matching probability rates is very much possible for hardware, though the determination of software failure rates or attack rates can only be a good estimate. Thus, the quality of the results can merely be as good as the input to the model, making the assumption of worst-case estimates as inputs the safest application. However, this only applies if the goal is to identify definite results. Alternatively, the methodology can be used to compare different architectures in the development process, i.e. given two different dependency graphs modelling the same components with the same specification, which of them is more robust with respect to potential failures and attacks? Based on the achieved results the most suitable architecture between multiple concurrent dependency graphs can be chosen, or a modification of the desired dependency graph can be performed and the analysis revisited. Another option could be to use the analysis to identify the worst-case failure and attack rates of a component, so that it still satisfies the requirements for the system's safety. On behalf of these results, the concrete component to be used in production may be chosen. For instance, one manufacturer may promise a lower failure rate RADAR than another.

9.2 Challenges

The strongest limitation of this research remains the lack of information on the precise internals of vehicular architectures. Due to the ever-ongoing competition between vehicle manufacturers but also between automotive part suppliers, this information is largely treated as confidential. While it is understandable that this knowledge is protected, and can even foster greater innovation [JSH14] through scarcity, this approach constrains advancements in fields reliant on large amounts of varied data such as in the training of AI models. Especially regarding security the confidentiality of solutions can become obscure. The OBD-II interface makes a great example. Recalling, this interface allows direct access to the internal vehicle network and the possibility of installing software on connected ECUs [Amm+20]. With growing connectivity, remote access to it via, e.g., OBD-II dongles is increasing. While the access to this interface is protected with an authentication scheme standardised by ISO 15031-7 [13] (SAE J2186), it does not follow current cybersecurity guidelines: As pointed out in [Amm+20], a simple challenge-response protocol (the *seed-key protocol*) on the basis of pre-shared keys is used which is cryptographically out-dated and the protocol itself is not standardised. Further, key length or random number generators are not specified by the prevailing standard. CAN bus tables specifying the codification of the messages specific to the car model are kept secret, seemingly as an attempt at protection by manufacturers. This idea is, however, projecting a false sense of security as many research publications in the past have shown how easily these messages can be reverse engineered [Kan+18; MS19]. In conclusion, a less restricted collaboration of research and industry in automotive would constitute in finding better and faster solutions for safety and security issues.

9.3 Extended Application

While the automotive domain served as an inspiration for this work, it must be mentioned that the methodology is not necessarily bound to it. Thus it can be applied in any other fields where the system that shall be modelled contains similar characteristics and has a comparable evaluation goal. Abstractly, this makes it applicable to any system or device that contains components of varying criticality in regard to maintaining its core task, functional data dependencies, possibly redundancies and access relationships. With the increasing connectivity of every day devices, application possibilities are becoming vast. Albeit every day smart devices are not highly critical systems, analysing the entry of safety- and security-wise failure states can still be beneficial for several reasons such as determining the

legal warranty provided by the manufacturer. As an example, we view the dependency graph of a smart coffee maker in Figure 9.1. The critical nodes here simply

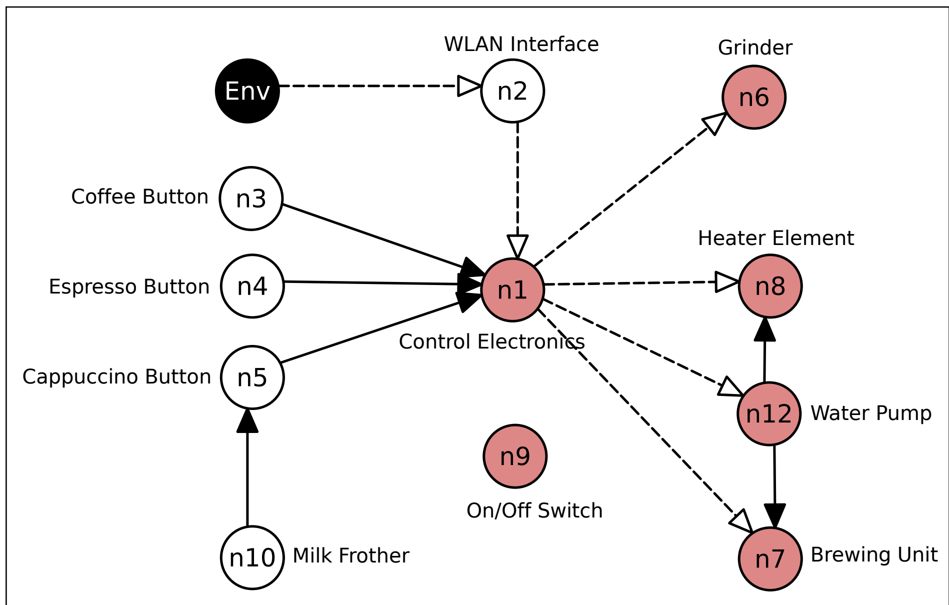


Figure 9.1: Dependency Graph of a Smart Coffee Maker

relate to all components that are mandatory to perform its main task – producing hot coffee. Various functional relationships can be modelled, for instance, the cappuccino program requiring the milk frother to work correctly. Some functional relationships can be omitted, like the On/Off Switch being important to any component, which is already provided by modelling it critical. Node redundancy can be used for the coffee program buttons to express that at least one program must still function so that the coffee maker is able to fulfil its task, although in a pretty degraded manner. Since this coffee maker is a smart home device, it provides a WLAN interface which makes it accessible from the outside, to e.g. trigger the coffee making process remotely. Therefore, an Env-node with Reach-dependencies is modelled which could potentially be abused by an attacker. This example shows that the dependency graph is abstract and flexible enough, so that also rather small and uncritical systems, independent from the automotive domain, can be modelled with it.

However, apart from describing technical systems and devices the initial meaning of the methodology could be stretched further to model different kinds of systems such as systems of an actor-based nature. Therefore the definition of a system component must be rethought. Previously, a node modelled a technical component, usually in terms of an enclosed system, though we also saw that a node can relate

to a software application (e.g. the firewall node). In the same manner, functional data dependencies can be used to model the relationship of multiple software applications running on hardware devices. Therewith independent failure and attack rates of single software applications can be modelled, though, once its operational core fails the application instantly ceases to operate as well. This shows a certain flexibility in the node definition. In actor-based systems a node would refer to an abstract participant or entity which can represent multiple actors of a category, rather than a specific mechanical or electronic part. When relating to multiple actors, failure and attack rates must be comprised accordingly. The criticality of nodes and the resulting Fail-state definition can for one depend on the importance of the actor to the system, and for other and more interestingly, on the perspective of a defined actor. The first is comparable to the standard system Fail-state where the purpose of the system cannot be provided anymore and the latter to system states that are undesirable for a certain actor.

To illustrate this idea, Figure 9.2 shows the dependency graph of a fictive payment system in an online shop. Vendors can provide products via the webshop and customers can place orders on these. For payment transactions, the webshop forwards the transaction details to the bank of the customer that performs the transaction with the bank of the vendor.

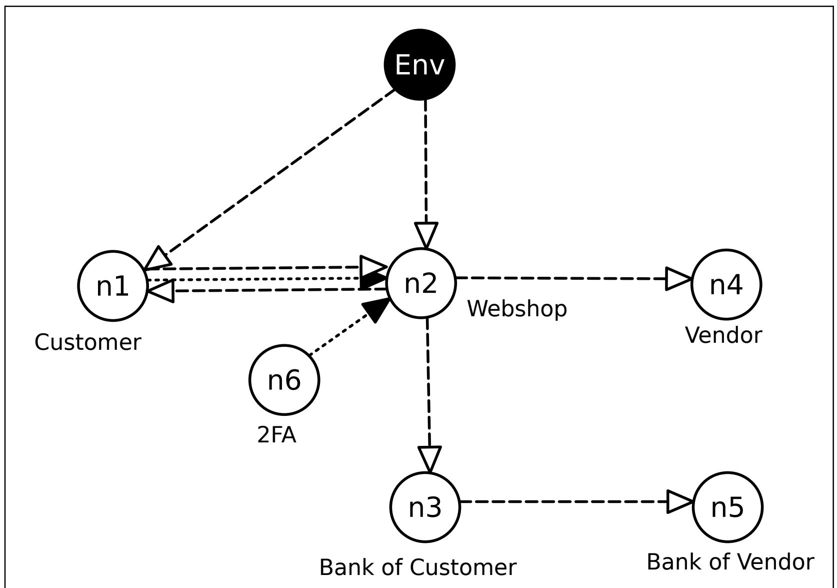


Figure 9.2: Actor-based Dependency Graph of a Payment System

The potential customer is modelled as node n_1 that employs a Reach-dependency to the webshop n_2 to place orders and so on. Further Reach-links are modelled to

express the described interaction between the webshop, the vendor and the banks. Since both the customer as well as the webshop are connected to the internet via some connected computers, they are reachable via the Env-node. Thereby a Two Factor Authentication (2FA) premise for logging in at the webshop is enhancing its protection. In addition to that, the Sec-link from the customer to the webshop is exploited in order to model an interesting behaviour: The direct attack on the webshop is less probable than the attack via a registered user (the customer). With that we can model that the attacker impersonating the customer is more successful in manipulating the webshop than a direct attack. In the system analysis, successful attacks on the modelled actors but also potential failures, which could relate to outages of webshop's servers, can be considered. Moreover, even a problem with the 2FA implementation could be concerned. As described, the evaluation goal dictated by the system's criticality definition depends on the person of view. For the general system failure, all nodes except for the 2FA node would be critical. However, if we choose the perspective of the customer to, for instance, analyse the probability that he falls victim of fraud, we would need to model his bank n_3 as critical.

By adjusting the meaning of the evaluation goal in dependence to the modelled system it becomes possible to evaluate further system properties that have previously been neglected. For instance, privacy concerns could be regarded if we model a system where the corruption of a node translates to the loss of privacy. An example for that would be patient data that is stored at a health insurance company. This highly sensitive data must be accessible by various doctors, perhaps partially at the pharmacy and by the patient itself. Privacy and confidentiality concerns arise for the patient, though, also integrity plays a fundamental role in ensuring that the patient receives the correct treatment. Given that, the corruption of the node relates to the loss of the patient's privacy and depending on the system's internals, it could also mean that the data's correctness becomes questionable.

While these extended application possibilities look promising, further research must be conducted to guarantee a safe application to non-technical systems. For example, it must be studied whether a modularization of actor-based systems follows the same conditions as before. If so, the modularization scheme could perhaps be used to model the internals of an actor, e.g. the internal structure of the webshop or the bank. Furthermore, it is questionable if we can consider recovery in these kinds of systems: If the corruption of the node relates to stolen credentials, then a reset of these credentials could be a recovery. However, if the corruption of a node relates to the loss of privacy, the recovery would mean that attackers would lose their obtained information and that is highly unlikely.

Lastly, to justify this application and identify benefits but also downsides, the general expressibility and evaluation possibilities of existing actor models must be researched and compared to this method.

9.4 Future Work

The presented work shows a complete concept, yet several additions are conceivable. These can be separated into two main categories: (1) modelling accuracy and extensions and (2) evaluation and tool enhancements. The first category concerns features that enhance the dependency graph with the goal of reaching a more realistic system modelling. This category has to be handled with care, as adding more details to the dependency graph will most likely lead to a more complex quantitative Markov model by expanding the state and transition growth further. The latter category mostly consists of advancements of ERIS with the intention of making it more user-friendly and performant.

Regarding category (1), an addition to consider the effects of self-protecting measures could be made. As described in Chapter 6, several countermeasures against security attacks include a proactive step of changing the system configuration or communication properties. For example, a fast and effective solution is the isolation of a successfully attacked component or a component group to limit negative effects and prevent the attack from spreading, by, e.g., stopping (ignoring) the communication with the affected component or shutting it off in a bus-like system. Depending on the task or communication position of the attacked component, it could be necessary to reconfigure the communication structure in order to maintain availability, e.g., in an internet-based context, the server is shut down and all requests are redirected to the back up server (see also [YEM14]). A simple possibility to add such a protective behaviour to the current way of modelling would be the dynamic removal of Reach-links to stop an attack from propagating. Yet, Reach-links do not reflect the communication structure precisely, as they are merely a representation of any existing access-relationships. Thus, information on the actual connection is abstracted away: Which components are using the same communication system? Is the connection direct or routed via several aligned communication systems? Furthermore, a communication cut-off would also mean that existing functional dependencies would have to be neglected. Consequently, further work is required to model the communication structure more closely, e.g., as in defining a specific type of node for it. Another promising advancement could be the consideration of failures that fall into the field of SOTIF [22]. Since these kind of failures or attacks have a more temporary nature since

they are not the failure of the component itself, but the surroundings leading to an undesired behaviour, they do not fit into the current health state definition. For instance, modelling them as a temporary change into the defective state would clash with existing recovery transitions: In the Markov model, these rates would be summed up, but a recovery from a functional failure such as a restart will not fix a sensor's obscuration. Given that, extra states are mandatory which, unavoidably, increase the state space again. Moreover, considering these failures may not make much sense when the evaluation goal is set to the entire lifetime of the system.

Concerning category (2), despite enhancing the general usability, further automations as in automatically determining modularization candidates and the generation of further CSL properties could be established. This would advance ERIS in becoming more detached from the PRISM usage. In order to determine modularization candidates automatically, rules for good and bad modularizations would have to be defined and implemented. The basis for that has already been established by the heuristics in Section 5.3. On behalf of that, the system model would have to be parsed and potential subgraphs identified. The difficulty then lies in determining which subgraphs and also the amount of subgraphs that are made into modules, while considering the trade-off between state space reduction and loss of accuracy. Luckily graph partitioning is a widely researched field and thus existing algorithms could potentially be reviewed and applied. Thereby, it would be beneficial to also take input rates into account. However, this requires further research in that area. After that, the next step would be to automatically choose the most suiting evaluation approach for the module. Though, since currently simulation models cannot be generated by dependency graphs, the algorithm would require additional information.

Author’s Contributions

[Hor+22] Timo F. Horeis, Fabian Plinke, Tobias Kain, Hans Tompits, Rhea C. Rinaldo and Johannes Heinrich. ‘Ein industrieübergreifender Überblick von fehlertoleranten Ansätzen in autonomen Systemen’. In: *Fahrerassistenzsysteme und automatisiertes Fahren*. Ed. by VDI Wissensforum GmbH. Vol. 2394. VDI-Berichte. 2022, pp. 89–108. ISBN: 978-3-18-102394-5. DOI: 10.51202/9783181023945.

[HR23] Timo Frederik Horeis and Rhea C. Rinaldo. ‘Towards Verification of Self-Healing for Autonomous Vehicles’. In: *33rd European Safety and Reliability Conference*. Ed. by Mário P. Brito, Terje Aven, Piero Baraldi, Marko Čepin and Enrico Zio. Research Publishing, 2023. DOI: 10.3850/978-981-18-8071-1_P122-cd.

[RH20a] Rhea C. Rinaldo and Timo F. Horeis. ‘A Hybrid Model for Safety and Security Assessment of Autonomous Vehicles’. In: *Computer Science in Cars Symposium*. CSCS ’20. Feldkirchen, Germany: Association for Computing Machinery, 2020. ISBN: 9781450376211. DOI: 10.1145/3385958.3430478.

[RH20b] Rhea C. Rinaldo and Dieter Hutter. ‘Integrated Analysis of Safety and Security Hazards in Automotive Systems’. In: *Computer Security*. Ed. by Sokratis K. Katsikas and Frederic Cuppens. Vol. 12501. Lecture Notes in Computer Science. ESORICS 2020, workshop CyberICPS. Berlin, Heidelberg: Springer, 2020, pp. 3–18. ISBN: 978-3-030-64329-4. DOI: 10.1007/978-3-030-64330-0_1.

[RH22] Rhea C. Rinaldo and Dieter Hutter. ‘Dependency Graph Modularization for a Scalable Safety and Security Analysis’. In: *32nd European Safety and Reliability Conference*. Ed. by Maria Chiara Leva, Edoardo Patelli, Luca Podofillini and Simon Wilson. Research Publishing, 2022. DOI: 10.3850/978-981-18-5183-4_R26-05-642-cd.

[RHK21] Rhea C. Rinaldo, Timo F. Horeis and Tobias Kain. 'Hybrid Modeling for the Assessment of Complex Autonomous Systems - A Safety and Security Case Study'. In: *Proceedings of the 31th European Safety and Reliability Conference*. Ed. by Bruno Castanier, Marko Cepin, David Bigaud and Christophe Berenguer. Vol. 31. Research Publishing, 2021. ISBN: 978-981-18-2016-8. DOI: 10.3850/978-981-18-2016-8_519-cd.

Abbreviation

2FA Two Factor Authentication

ABS Anti-lock Braking System
AD Autonomous Driving
AI Artificial Intelligence
ALKS Automated Lane Keeping System
ASIL Automotive Safety Integrity Level

BDMP Boolean logic Driven Markov Processes

CAN Controller Area Network
CIA Confidentiality, Integrity and Availability
CSL Continuous Stochastic Logic
CSV Comma-separated Values
CTMC Continuous-Time Markov Chain
CVE Common Vulnerabilities and Exposures
CVSS Common Vulnerability Scoring System

DMR Dual Modular Redundancy
DOS Denial of Service
DTMC Discrete-Time Markov Chain

ECC Error Correcting Code
ECU Electronic Control Unit
EHW Evolvable Hardware
EmHW Embryonic Hardware

FDIR Fault Detection Isolation and Recovery
FPGA Field Programmable Gate Array
FTA Fault Tree Analysis

GNSS Global Navigation Satellite System
GPL GNU Public License

GPS Global Positioning System
GSM Global System for Mobile Communications
GUI Graphical User Interface

HARA Hazard Analysis and Risk Assessment
HMI Human Machine Interface
HSM Hardware Security Module

IDS Intrusion Detection System
IMC Interactive Markov Chains

JSON JavaScript Object Notation

KPI Key Performance Indicator

LDWS Lane Departure Warning System
LIDAR Light Detection And Ranging
LIN Local Interconnect Network

MAC Message Authentication Code
MCS Monte Carlo Simulation
MDP Markov Decision Process
MOST Media Oriented Systems Transport
MTBF Mean Time Between Failures
MTTF Mean Time To Failure
MTTR Mean Time To Repair
MTTSF Mean Time To Security Failure

NHTSA National Highway Traffic Safety Administration

OBD-II On-Board Diagnostic II
OC Organic Computing
ODD Operational Design Domain

PCTL Probabilistic Computation Tree Logic
PDF Portable Document Format
PEPA Performance Evaluation Process Algebra

RADAR Radio Detection And Ranging
RBD Reliability Block Diagram

ROS	Robot Operating System
SAE	Society of Automotive Engineers
SDS	Self Driving System
SIL	Safety Integrity Level
SOTIF	Safety Of The Intended Functionality
TARA	Threat Analysis and Risk Assessment
TMR	Triple Modular Redundancy
USB	Universal Serial Bus
V2X	Vehicle-to-X
VANET	Vehicular ad hoc Network
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

List of Tables

1.1 Outline . . . 12

2.1 Vehicular Communication Systems . . . 18

7.1 Model Sizes of the Different Modularization Options . . . 125
7.2 Assumed Failure and Attack Rates . . . 125

8.1 Results of the Comparison . . . 133

B.1 Example 1: Result Probabilities and Module Rates at Time T . . . 187
B.2 Example 2: Result Probabilities and Module Rates at Time T . . . 188
B.3 Result Probabilities and Input Rates at Time T . . . 189
B.4 AT-CARS Module Rates . . . 190
B.5 Option 3 Evaluation with AT-CARS Module . . . 190
B.6 AT-CARS Example Iteration . . . 191
B.7 Corruption Result Probabilities . . . 192

List of Figures

2.1 Levels of Driving Automation by SAE J3016 [21c; SAE21] 15
2.2 Modern Vehicle Architecture with SDS 17
2.3 Autoware Software Stack [Kat+18] 20
2.4 Boss Sensor Placement [UA+09] 24
2.5 Essential Steps of a Safety/Security Risk Assessment (simplified) . . 29
2.6 Assessment Paradigms Overview 35
2.7 Monte-Carlo Simulation Concept 41

4.1 Motivational Example of a Dependency Graph 52
4.2 Security Function Example 54
4.3 Nodes with Different Criticality 56
4.4 Dependency Graph of an Autonomous Vehicle 57
4.5 Conceptual Illustration of the $\mathcal{DMC}$ 61
4.6 Toy Example of a Dependency Graph 63
4.7 $\mathcal{DMC}$ of the Toy Example: Multiple Failure States 64
4.8 $\mathcal{DMC}$ of the Toy Example: Comprised Failure States 64
4.9 $\mathcal{DMC}$ Evaluation Example 67
4.10 Plot: Property P_1 over 12 Months of Continuous Operation 68

5.1 Modularization Example 73
5.2 Modularization with Functional Dependencies 75
5.3 Non-modularizable Functional Dependencies 76
5.4 Analytical Evaluation Procedure in a Modularized System 79
5.5 Hybrid Evaluation Procedure in a Modularized System 82
5.6 Example Path through Modularization 84
5.7 Imaginary Path with Defects and Corruptions 85
5.8 Problematic Security (Reach-path) Modularization 86
5.9 Plot: Problematic Reach-path Modularization 88
5.10 Problematic Security (Reach-link) Modularization 88
5.11 Plot: Problematic Reach-link Modularization 89
5.12 Depth-2 Modularization 90

6.1 General Recovery Scheme 98

6.2 Recovery Example System 100
6.3 Plot: Comparison of Recovery Strategies 101

7.1 ERIS Main Window 104
7.2 Experiment Settings Window 107
7.3 ERIS Node Settings Window 109
7.4 Node Dependency Example 110
7.5 Transformation Example System 112
7.6 ERIS Module Node 118
7.7 Module Evaluation Process 120
7.8 Evaluation Process of a Non-modularized System 121
7.9 Modularization Options of Autonomous Vehicle Dependency Graph 124
7.10 Plot: Option 3 Evaluation Results with AT-CARS Module 126
7.11 Plot: Option 3 Corruption Probabilities for the Sensitivity Analysis 127

8.1 Small Example for Comparison 132

9.1 Dependency Graph of a Smart Coffee Maker 138
9.2 Actor-based Dependency Graph of a Payment System 139

Bibliography

[13] *Road vehicles – Communication between vehicle and external equipment for emissions-related diagnostics — Part 7: Data link security. Second edition.* Standard ISO 15031-7. Geneva, Switzerland: International Organization for Standardization (ISO), 2013.

[18] *Road vehicles – Functional safety. Second edition.* Standard ISO 26262. Geneva, Switzerland: International Organization for Standardization (ISO), 2018.

[21a] *Abbendum 156 – UN Regulation No. 157. Uniform provisions concerning the approval of vehicles with regard to Automated Lane Keeping Systems.* Agreement. United Nations, 2021.

[21b] *Road vehicles – Cybersecurity engineering. First edition.* Standard ISO/SAE 21434. Geneva, Switzerland and Warrendale, USA: International Organization for Standardization (ISO) and Society of Automotive Engineers (SAE) International, 2021.

[21c] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles.* Standard SAE J3016. USA and Switzerland: Society of Automotive Engineers (SAE) International, 2021.

[22] *Road vehicles – Safety of the intended functionality.* Standard ISO 21448:2022. Geneva, Switzerland: International Organization for Standardization (ISO), 2022.

[Adm08] National Highway Traffic Safety Administration. 'National motor vehicle crash causation survey: Report to congress'. In: *National Highway Traffic Safety Administration Technical Report DOT HS* 811 (2008), p. 59.

[Adr+15] David Adrian et al. 'Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice'. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 5–17. ISBN: 9781450338325. DOI: 10.1145/2810103.2813707.

[Al-+19] Omar Y. Al-Jarrah et al. 'Intrusion Detection Systems for Intra-Vehicle Networks: A Review'. In: *IEEE Access* 7 (2019), pp. 21266–21289. DOI: 10.1109/ACCESS.2019.2894183.

[ALL18] Hooshyar Azizpour, Mary Lundteigen and Yiliu Liu. 'Analysis of Simplification in Markov Based Approach for Performance Assessment of Safety Instrumented System (SIS)'. In: *Reliability Engineering & System Safety* 183 (2018). DOI: 10.1016/j.ress.2018.09.012.

[Alt95] Thomas J. Altenbach. 'A Comparison of Risk Assessment Techniques from Qualitative to Quantitative'. In: Honululu, Hawaii: Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 1995.

[Amm+20] Mahmoud Ammar et al. 'Securing the On-board Diagnostics Port (OBD-II) in Vehicles'. In: *SAE International Journal of Transportation Cybersecurity and Privacy* 11-02-02-0009 (2020), pp. 83–106. DOI: 10.4271/11-02-02-0009.

[ANM14] Tanzirul Azim, Iulian Neamtiu and Lisa M. Marvel. 'Towards self-healing smartphone software via automated patching'. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 623–628. DOI: 10.1145/2642937.2642955.

[AR06] Konstantin Andreev and Harald Racke. 'Balanced Graph Partitioning'. In: *Theory of Computing Systems* 39 (2006), pp. 929–939. DOI: 10.1007/s00224-006-1350-7.

[ASB07] Matthias Althoff, Olaf Stursberg and Martin Buss. 'Safety Assessment of Autonomous Cars using Verification Techniques'. In: *2007 American Control Conference*. IEEE. 2007, pp. 4154–4159. DOI: 10.1109/ACC.2007.4282809.

[Azi+96] Adnan Aziz et al. 'Verifying Continuous Time Markov Chains'. In: *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*. Ed. by R. Alur and T. Henzinger. Vol. 1102. LNCS. Springer, 1996, pp. 269–276. DOI: 10.1007/3-540-61474-5_75.

[BS13] Charles-Edmond Bichot and Patrick Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[Bul+16] Aydın Buluç et al. 'Recent Advances in Graph Partitioning'. In: *Algorithm Engineering: Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Cham: Springer International Publishing, 2016, pp. 117–158. ISBN: 978-3-319-49487-6. DOI: 10.1007/978-3-319-49487-6_4.

[Car+08] João B. Cardoso et al. 'Structural reliability analysis using Monte Carlo simulation and neural networks'. In: *Advances in Engineering Software* 39.6 (2008), pp. 505–513. ISSN: 0965-9978. DOI: 10.1016/j.advengsoft.2007.03.015.

[CF01] George Candea and Armando Fox. 'Recursive restartability: turning the reboot sledgehammer into a scalpel'. In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. 2001, pp. 125–130. DOI: 10.1109/HOTOS.2001.990072.

[Cru23] Cruise. *Important Updates from Cruise*. 2023. URL: https://getcruise.com/news/blog/2023/important-updates-from-cruise/ (visited on 3rd Dec. 2023).

[Cui+19] Jin Cui et al. 'Collaborative Analysis Framework of Safety and Security for Autonomous Vehicles'. In: *IEEE Access* 7 (2019), pp. 148672–148683. DOI: 10.1109/ACCESS.2019.2946632.

[Cyb] The European Union Agency for Cybersecurity (ENISA). *Vulnerabilities and Exploits*. URL: https://www.enisa.europa.eu/topics/incident-response/glossary/vulnerabilities-and-exploits (visited on 22nd Dec. 2022).

[Dek+05] Frederik Michel Dekking et al. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer, 2005. ISBN: 1-85233-896-2.

[DRU08] Michael Darms, Paul E. Rybski and Chris Urmson. 'A Multisensor Multiobject Tracking System for an Autonomous Vehicle Driving in an Urban Environment'. In: *Proceedings of AVEC 2008 - 9th International Symposium on Advanced Vehicle Control*. Kobe, 2008.

[DT16] Nabarun Das and William Taylor. 'Quantified Fault Tree Techniques for Calculating Hardware Fault Metrics According to ISO 26262'. In: 2016, pp. 1–8. DOI: 10.1109/ISPCE.2016.7492848.

[FS17] Sibylle Fröschle and Alexander Stühring. 'Analyzing the capabilities of the CAN attacker'. In: *European Symposium on Research in Computer Security*. Springer International Publishing, 2017, pp. 464–482. DOI: 10.1007/978-3-319-66402-6_27.

[Gla+15] Benjamin Glas et al. 'Automotive safety and security integration challenges'. In: *Automotive - Safety & Security 2014*. GI e.V., 2015, pp. 13–28.

[Gon22] Ivan Goncharov. *Autonomous Vehicle Companies And Their ML.* 2022. URL: https://wandb.ai/ivangoncharov/AVs-report/reports/Autonomous-Vehicle-Companies-And-Their-ML--VmlldzoyNTg1Mjc1 (visited on 13th Jan. 2023).

[Här+23] Ivo Häring et al. 'Overall Markov diagram design and simulation example for scalable safety analysis of autonomous vehicles'. In: *33rd European Safety and Reliability Conference*. Ed. by Mário P. Brito et al. Research Publishing, 2023. DOI: 10.3850/978-981-18-8071-1_P348-cd.

[Has+17] Hamssa Hasrouny et al. 'VANet security challenges and solutions: A survey'. In: *Vehicular Communications* 7 (2017), pp. 7–20. ISSN: 2214-2096. DOI: 10.1016/j.vehcom.2017.01.002.

[Hei+19] Johannes Heinrich et al. 'State-based Availability Analysis of Hard- and Software Architectures using Monte Carlo Simulation under Consideration of Different Failure modes and Degradation Models'. In: *Proceedings of the 29th European Safety and Reliability Conference (ESREL)*. Hannover, Germany: Research Publishing Services, 2019, pp. 1970–1978. DOI: 10.3850/978-981-11-2724-3_0333-cd.

[Hei23] Martin Holland (Heise). *Unfall mit Personenschaden: Cruise verliert Erlaubnis für Betrieb von Robotaxis*. 2023. URL: https://www.heise.de/news/Unfall-mit-Personenschaden-Cruise-verliert-Erlaubnis-fuer-Betrieb-von-Robotaxis-9343410.html (visited on 3rd Dec. 2023).

[HJ95] Hans Hansson and Bengt Jonsson. 'A Logic for Reasoning about Time and Reliability'. In: *Formal Aspects of Computing* 6 (1995). DOI: 10.1007/BF01211866.

[HKK13] Holger Hermanns, Jan Krčál and Jan Křetínský. 'Compositional Verification and Optimization of Interactive Markov Chains'. In: *CONCUR 2013 – Concurrency Theory*. Ed. by Pedro R. D'Argenio and Hernán Melgratti. Berlin, Heidelberg: Springer, 2013, pp. 364–379. ISBN: 978-3-642-40184-8. DOI: 10.1007/978-3-642-40184-8_26.

[Hor+22] Timo F. Horeis et al. 'Ein industrieübergreifender Überblick von fehlertoleranten Ansätzen in autonomen Systemen'. In: *Fahrerassistenzsysteme und automatisiertes Fahren*. Ed. by VDI Wissensforum GmbH. Vol. 2394. VDI-Berichte. 2022, pp. 89–108. ISBN: 978-3-18-102394-5. DOI: 10.51202/9783181023945.

[HR23] Timo Frederik Horeis and Rhea C. Rinaldo. 'Towards Verification of Self-Healing for Autonomous Vehicles'. In: *33rd European Safety and Reliability Conference*. Ed. by Mário P. Brito et al. Research Publishing, 2023. DOI: 10.3850/978-981-18-8071-1_P122-cd.

[JSH14] Ruey-Jer "Bryan" Jean, Rudolf R. Sinkovics and Thomas P. Hiebaum. 'The Effects of Supplier Involvement and Knowledge Protection on Product Innovation in Customer–Supplier Relationships: A Study of Global Automotive Suppliers in China'. In: *Journal of Product Innovation Management* 31.1 (2014), pp. 98–113. DOI: 10.1111/jpim.12082.

[Kai+20] Tobias Kain et al. 'FDIRO: A General Approach for a Fail-Operational System Design'. In: *Proceedings of the 30th European Safety and Reliability Conference and 15th Probabilistic Safety Assessment and Management Conference*. Ed. by Francesco Di Maio Piero Baraldi and Enrico Zio. Research Publishing, 2020. ISBN: 978-981-14-8593-0. DOI: 10.3850/978-981-14-8593-0.

[Kan+18] Tae Un Kang et al. 'Automated Reverse Engineering and Attack for CAN Using OBD-II'. In: *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*. 2018, pp. 1–7. DOI: 10.1109/VTCFall.2018.8690781.

[Kat+07] Joost Pieter Katoen et al. 'Three-Valued Abstraction for Continuous-Time Markov Chains'. In: *Proceedings of the 19th International Conference on Computer Aided Verification*. CAV'07. Berlin, Heidelberg: Springer, 2007, pp. 311–324. ISBN: 9783540733676. DOI: 10.1007/978-3-540-73368-3_37.

[Kat+18] Shinpei Kato et al. 'Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems'. In: *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018*. 2018, pp. 287–296. DOI: 10.1109/ICCPS.2018.00035.

[Kat13] Joost-Pieter Katoen. 'Model Checking Meets Probability: A Gentle Introduction'. In: 2013, p. 29. DOI: 10.3233/978-1-61499-207-3-177.

[Kat16] Joost-Pieter Katoen. 'The Probabilistic Model Checking Landscape'. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 31–45. ISBN: 9781450343916. DOI: 10.1145/2933575.2934574.

[KC13] Igor Kotenko and Andrey Chechulin. 'A Cyber Attack Modeling and Impact Assessment Framework'. In: *5th International Conference on Cyber Conflict*. 2013.

[Ken21] Will Kenton. *Monte Carlo Simulation*. 2021. URL: https://www.investopedia.com/terms/m/montecarlosimulation.asp (visited on 21st June 2022).

[KES09] Hyun Gook Kang, Heung-Seop Eom and Han Seong Son. 'Software failure probability quantification for system risk assessment'. In: *Scholarly Research Exchange* 2009 (2009). DOI: 10.3814/2009/163456.

[Kha+19] Kasem Khalil et al. 'Self-healing hardware systems: A review'. In: *Microelectronics Journal* 93 (2019), p. 104620. ISSN: 0026-2692. DOI: 10.1016/j.mejo.2019.104620.

[KNP10] Marta Kwiatkowska, Gethin Norman and David Parker. 'Advances and Challenges of Probabilistic Model Checking'. In: *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE Press, 2010, pp. 1691–1698. DOI: 10.1109/ALLERTON.2010.5707120.

[KNP11] Marta Kwiatkowska, Gethin Norman and David Parker. 'PRISM 4.0: Verification of Probabilistic Real-time Systems'. In: *23rd International Conference on Computer Aided Verification (CAV'11)*. Vol. 6806. LNCS. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_47.

[Kos+10] Karl Koscher et al. 'Experimental Security Analysis of a Modern Automobile'. In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 447–462. DOI: 10.1109/SP.2010.34.

[Kri+15] Siwar Kriaa et al. 'A survey of approaches combining safety and security for industrial control systems'. In: *Reliability engineering & system safety* 139 (2015), pp. 156–178. DOI: 10.1016/j.ress.2015.02.008.

[KS17] Rajesh Kumar and Mariëlle Stoelinga. 'Quantitative Security and Safety Analysis with Attack-Fault Trees'. In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. 2017, pp. 25–32. DOI: 10.1109/HASE.2017.12.

[KWS21] Florian Kaiser, Marcus Wiens and Frank Schultmann. 'Motivation-based Attacker Modelling for Cyber Risk Management: A Quantitative Content: Analysis and a Natural Experiment'. In: *Journal of Information Security and Cybercrimes Research* 4 (2021), pp. 30–43. DOI: 10.26735/NMMD9869.

[LA21] Michael Lee and Travis Atkison. 'VANET applications: Past, present, and future'. In: *Vehicular Communications* 28 (2021), p. 100310. ISSN: 2214-2096. DOI: 10.1016/j.vehcom.2020.100310.

[Lin+06] Maria B. Line et al. 'Safety vs. Security?' In: *Proceedings of the 8th International Conference on Probabilistic Safety Assessment & Management (PSAM)*. ASME Press, 2006. ISBN: 0791802442. DOI: 10.1115/1.802442.paper151.

[Lon+19] Stefano Longari et al. 'A Secure-by-Design Framework for Automotive On-board Network Risk Analysis'. In: *2019 IEEE Vehicular Networking Conference (VNC)*. 2019, pp. 1–8. DOI: 10.1109/VNC48660.2019.9062783.

[LR16] Fan Long and Martin Rinard. 'Automatic Patch Generation by Learning Correct Code'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery (ACM), 2016, pp. 298–312. ISBN: 9781450335492. DOI: 10.1145/2837614.2837617.

[Mad+02] Bharat B. Madan et al. 'Modeling and Quantification of Security Attributes of Software Systems'. In: *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. DSN '02. USA: IEEE Computer Society, 2002, pp. 505–514. ISBN: 0769515975. DOI: 10.1109/DSN.2002.1028941.

[Mis08] Krishna B. Misra. *Handbook of Performability Engineering*. Springer, 2008, p. 1316. ISBN: 978-1-84800-130-5. DOI: 10.1007/978-1-84800-131-2.

[Mon18] Martin Monperrus. 'Automatic Software Repair'. In: *ACM Computing Surveys* 51.1 (2018), pp. 1–24. ISSN: 0360-0300. DOI: 10.1145/3105906.

[MP10] Arno Meyna and Bernhard Pauli. *Zuverlässigkeitstechnik: Quantitative Bewertungsverfahren*. Hanser Verlag, 2010.

[MS01] Adamantios Mettas and Manolis Savva. 'System reliability analysis: the advantages of using analytical methods to analyze non-repairable systems'. In: *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity*. Philadelphia, PA, USA: IEEE, 2001, pp. 80–85. DOI: 10.1109/RAMS.2001.902446.

[MS19] Mirco Marchetti and Dario Stabili. 'READ: Reverse Engineering of Automotive Data Frames'. In: *IEEE Transactions on Information Forensics and Security* 14.4 (2019), pp. 1083–1097. DOI: 10.1109/TIFS.2018.2870826.

[Mun+15] Philipp Mundhenk et al. 'Security analysis of automotive architectures using probabilistic model checking'. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744906.

[MV14] Charlie Miller and Chris Valasek. 'A Survey of Remote Automotive Attack Surfaces'. In: *Black Hat USA* 2014 (2014), p. 94.

[MV15] Charlie Miller and Chris Valasek. 'Remote Exploitation of an Unaltered Passenger Vehicle'. In: (2015), pp. 1–91. URL: http://illmatics.com/Remote%20Car%20Hacking.pdf (visited on 3rd Dec. 2019).

[Nor] TÜV Nord. *Fahrassistenzsysteme - Pflicht ab dem 6. Juli 2022*. URL: https://www.tuev-nord.de/de/privatkunden/ratgeber-und-tipps/technik/fahrassistenzsysteme/ (visited on 13th Jan. 2023).

[NR16] Stefan Nürnberger and Christian Rossow. '–vatiCAN– Vetted, Authenticated CAN Bus'. In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Berlin, Heidelberg: Springer, 2016, pp. 106–124. ISBN: 978-3-662-53140-2. DOI: 10.1007/978-3-662-53140-2_6.

[OC91] John O'Quigley and Sylvie Chevret. 'Methods for dose finding studies in cancer clinical trials: A review and results of a monte carlo study'. In: *Statistics in Medicine* 10 (1991), pp. 1647–1664. DOI: 10.1002/sim.4780101104.

[OT00] Cesar Ortega-Sanchez and Andy Tyrrell. 'A Hardware Implementation of an Embryonic Architecture Using Virtex® FPGAs'. In: vol. 1801. 2000, pp. 155–164. ISBN: 978-3-540-67338-5. DOI: 10.1007/3-540-46406-9_16.

[Pap84] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 1984.

[PB10] Ludovic Piètre-Cambacédès and Marc Bouissou. 'Modeling safety and security interdependencies with BDMP (Boolean logic Driven Markov Processes)'. In: *2010 IEEE International Conference on Systems, Man and Cybernetics*. 2010, pp. 2852–2861. DOI: 10.1109/ICSMC.2010.5641922.

[PD11] Harald Psaier and Schahram Dustdar. ‘A Survey on Self-Healing Systems: Approaches and Systems’. In: *Computing* 91.1 (2011), pp. 43–73. ISSN: 0010-485X. DOI: 10.1007/s00607-010-0107-y.

[Pen09] Jiafei Peng. *Grobe Einführung der Fahrzeugelektronik und Vernetzungssystem*. 2009. URL: https://de.slideshare.net/Catsh/grobe-einfhrung-der-fahrzeugelektronik-und-vernetzungssystem (visited on 15th Jan. 2021).

[Per22] Fabian Pertschy. *Autonomes Fahren*. 2022. URL: https://www.automotiveit.eu/technology/autonomes-fahren/welcher-autobauer-hat-beim-autonomen-fahren-die-nase-vorn-124.html (visited on 13th Jan. 2023).

[Ray08] Samik Raychaudhuri. ‘Introduction to Monte Carlo simulation’. In: 2008, pp. 91–100. DOI: 10.1109/WSC.2008.4736059.

[RH20a] Rhea C. Rinaldo and Timo F. Horeis. ‘A Hybrid Model for Safety and Security Assessment of Autonomous Vehicles’. In: *Computer Science in Cars Symposium*. CSCS ’20. Feldkirchen, Germany: Association for Computing Machinery, 2020. ISBN: 9781450376211. DOI: 10.1145/3385958.3430478.

[RH20b] Rhea C. Rinaldo and Dieter Hutter. ‘Integrated Analysis of Safety and Security Hazards in Automotive Systems’. In: *Computer Security*. Ed. by Sokratis K. Katsikas and Frederic Cuppens. Vol. 12501. Lecture Notes in Computer Science. ESORICS 2020, workshop CyberICPS. Berlin, Heidelberg: Springer, 2020, pp. 3–18. ISBN: 978-3-030-64329-4. DOI: 10.1007/978-3-030-64330-0_1.

[RH22] Rhea C. Rinaldo and Dieter Hutter. ‘Dependency Graph Modularization for a Scalable Safety and Security Analysis’. In: *32nd European Safety and Reliability Conference*. Ed. by Maria Chiara Leva et al. Research Publishing, 2022. DOI: 10.3850/978-981-18-5183-4_R26-05-642-cd.

[RHK21] Rhea C. Rinaldo, Timo F. Horeis and Tobias Kain. ‘Hybrid Modeling for the Assessment of Complex Autonomous Systems - A Safety and Security Case Study’. In: *Proceedings of the 31th European Safety and Reliability Conference*. Ed. by Bruno Castanier et al. Vol. 31. Research Publishing, 2021. ISBN: 978-981-18-2016-8. DOI: 10.3850/978-981-18-2016-8_519-cd.

[Röd17] Jürgen Röder. *Automotive Ethernet - Die Zukunft der vernetzten Fahrzeugarchitektur*. 2017. URL: https://www.vdi-wissensforum.de/news/automotive-ethernet/ (visited on 3rd Aug. 2022).

[SAE21] SAE. *SAE Levels of Driving Automation™ Refined for Clarity and International Audience*. 2021. URL: https://www.sae.org/blog/sae-j3016-update (visited on 24th Feb. 2023).

[Sar94] Robert Sargent. 'A historical view of hybrid simulation/analytic models.' In: *Proceedings of Winter Simulation Conference*. Lake Buena Vista, FL, USA: IEEE, 1994, pp. 383–386. DOI: 10.1109/WSC.1994.717204.

[SB06] Mohammed Sharif and Donald H. Burn. 'Simulating climate change scenarios using an improved K-nearest neighbor model'. In: *Journal of Hydrology* 325.1 (2006), pp. 179–196. ISSN: 0022-1694. DOI: 10.1016/j.jhydrol.2005.10.015.

[SC05] Alexey Smirnov and Tzi-cker Chiueh. 'DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks.' In: *12th Annual Network and Distributed System Security Symposium*. 2005.

[Sch+17] Johannes Schlatow et al. 'Self-awareness in autonomous automotive systems'. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 2017, pp. 1050–1055. DOI: 10.23919/DATE.2017.7927145.

[Sch99] Bruce Schneier. 'Attack Trees'. In: *Dr. Dobb's Journal* 24.12 (1999), pp. 21–29.

[SDW14] Philipp Schleiss, Christian Drabek and Gereon Weiss. *SafeAdapt Deliverable 3.1 - Concept for Enforcing Safe Adaptation during Runtime*. 2014. URL: https://www.safeadapt.eu (visited on 26th Jan. 2023).

[She+21] Akhil Shetty et al. 'Safety challenges for autonomous vehicles in the absence of connectivity'. In: *Transportation Research Part C: Emerging Technologies* 128 (2021), p. 103133. ISSN: 0968-090X. DOI: 10.1016/j.trc.2021.103133.

[Smi10] Michael J. A. Smith. 'Compositional Abstraction of PEPA Models for Transient Analysis'. In: *Computer Performance Engineering*. Ed. by Alessandro Aldini et al. Springer Berlin Heidelberg, 2010, pp. 252–267. ISBN: 978-3-642-15784-4. DOI: 10.1007/978-3-642-15784-4_17.

[SS21] Sehajbir Singh and Baljit Singh Saini. 'Autonomous cars: Recent developments, challenges, and possible solutions'. In: *IOP Conference Series: Materials Science and Engineering* 1022 (2021), p. 012028. DOI: 10.1088/1757-899X/1022/1/012028.

[Sto+22] Torben Stolte et al. 'A taxonomy to unify fault tolerance regimes for automotive systems: defining fail- operational, fail-degraded, and fail-safe'. In: *IEEE Transactions on Intelligent Vehicles* 7.2 (2022), pp. 251–262. DOI: 10.1109/TIV.2021.3129933.

[TD21] Bipul Kumar Talukdar and Bimal Chandra Deka. 'An Approach to Reliability, Availability and Maintainability Analysis of a Plug-In Electric Vehicle'. In: *World Electric Vehicle Journal* 12.1 (2021). ISSN: 2032-6653. DOI: 10.3390/wevj12010034.

[Tor00] Wilfredo Torres-Pomales. *Software Fault Tolerance: A Tutorial.* National Aeronautics and Space Administration (NASA), 2000.

[TSM17] Sven Tomforde, Bernhard Sick and Christian Müller-Schloer. 'Organic Computing in the Spotlight'. In: *CoRR* abs/1701.08125 (2017). DOI: 10.48550/arXiv.1701.08125.

[UA+09] Chris Urmson, Joshua Anhalt et al. 'Autonomous Driving in Urban Environments: Boss and the Urban Challenge'. In: *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic.* Berlin, Heidelberg: Springer, 2009, pp. 1–59. ISBN: 978-3-642-03991-1. DOI: 10.1007/978-3-642-03991-1_1.

[Wal+23] Teo Puig Walz et al. 'Markov Modelling for Autonomous Vehicle Safety Assessment: Numerical Modularization to Avoid System State-Explosion'. In: *Proceedings of the 26th IEEE International Conference on Intelligent Transportation Systems.* Bilbao, Spain: IEEE, 2023.

[Wan+10] Lingyu Wang et al. 'k-Zero Day Safety: Measuring the Security Risk of Networks against Unknown Attacks'. In: *15th European Symposium on Research in Computer Security (ESORICS).* Vol. 6345. LNCS. Berlin, Heidelberg: Springer, 2010, pp. 573–587. DOI: 10.1007/978-3-642-15497-3_35.

[Wat19] Consumer Watchdog. *Kill Switch. Why connected Cars can be killing Machines and how to turn them off.* 2019. URL: https://consumerwatchdog.org/report/kill-switch-why-connected-cars-can-be-killing-machines-and-how-turn-them (visited on 3rd Sept. 2019).

[WCH11] Edward Wyrwas, Lloyd Condra and Avshalom Hava. 'Accurate Quantitative Physics-of-Failure Approach to Integrated Circuit Reliability'. In: *IPC APEX EXPO Technical Conference 2011* 3 (2011).

[WWP04] Marko Wolf, André Weimerskirch and Christof Paar. 'Security in Automotive Bus Systems'. In: *Workshop on Embedded Security in Cars (ESCAR)'04*. 2004, pp. 1–13.

[Yeh+17] Enoch R. Yeh et al. 'Security in Automotive Radar and Vehicular Networks'. In: *Microwave Journal* 60 (2017), pp. 148–164.

[YEM14] Eric Yuan, Naeem Esfahani and Sam Malek. 'A Systematic Survey of Self-Protecting Software Systems'. In: *ACM Trans. Auton. Adapt. Syst.* 8.4 (2014). ISSN: 1556-4665. DOI: 10.1145/2555611.

[You+06] Håkan L. S. Younes et al. 'Numerical vs. Statistical Probabilistic Model Checking'. In: *International Journal on Software Tools for Technology Transfer (STTT)* 8.3 (2006), pp. 216–228. DOI: 10.1007/s10009-005-0187-8.

[Zha+20] Lin Zhang et al. 'Real-Time Attack-Recovery for Cyber-Physical Systems Using Linear Approximations'. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 205–217. DOI: 10.1109/RTSS49844.2020.00028.

[Zio07] Enrico Zio. *An Introduction to the Basics of Reliability and Risk Analysis*. Vol. 13. World Scientific Publishing Company, 2007, p. 236. DOI: 10.1142/6442.

[Ziz05] Miroslav Zizka. 'The Analytic Approach Vs. The Simulation Approach to Determining Safety Stock'. In: *Problems and Perspectives in Management* 3 (2005).

A PRISM Code Listings

Subsequently, the complete ERIS-generated PRISM code of all examples evaluated in this thesis is provided.

A.1 Modularization

This section contains the ERIS-generated PRISM code used in Chapter 5. For the sake of clarity, subsections for each example system are established. Note that in cases where only the input values (failure and attack rates) and not the dependency graph have been changed, the code is only listed once.

A.1.1 Example 1: Problematic Security Modularization (Reach-path)

```
1  ctmc

3  const double rn1SEC = 0.2;
4  const double rn1SAFE = 0;
5  const double rn1GUAR = 0;
6  const double rn2SEC = 0.2;
7  const double rn2SAFE = 0;
8  const double rn2GUAR = 0;
9  const double rn3SEC = 0.2;
10 const double rn3SAFE = 0;
11 const double rn3GUAR = 0;
12 const double rn4SEC = 0.2;
13 const double rn4SAFE = 0;
14 const double rn4GUAR = 0;
15 const double rn5SEC = 0.2;
16 const double rn5SAFE = 0;
17 const double rn5GUAR = 0;
18 const double rn6SEC = 0.2;
19 const double rn6SAFE = 0;
20 const double rn6GUAR = 0;
```

```
21  const double rn7SEC = 0.2;
22  const double rn7SAFE = 0;
23  const double rn7GUAR = 0;

25  formula operational = (n4=0 | n7=0) & (n7=0 | n4=0) ;

27  module generatedScenario

29  n1: [0..2] init 0;
30  n2: [0..2] init 0;
31  n3: [0..2] init 0;
32  n4: [0..2] init 0;
33  n5: [0..2] init 0;
34  n6: [0..2] init 0;
35  n7: [0..2] init 0;

37  [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
38  [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
39  [] (n3=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
40  [] (n5=0) & (operational) -> rn5SEC : (n5'=2);
41  [] (n5=2 & n1=0) & (operational) -> rn1SEC : (n1'=2);
42  [] (n5=2 & n6=0) & (operational) -> rn6SEC : (n6'=2);
43  [] (n6=2 & n7=0) & (operational) -> rn7SEC : (n7'=2);

45  endmodule

47  label "systemfailure" = !operational;
48  label "defective" = (n4=1 & n7=1) | (n7=1 & n4=1) ;
49  label "corrupted" = (n4=2 & n7=2) | (n7=2 & n4=2) ;
```

Listing A.1: Example 1 Original Graph

```
1   ctmc

3   const double rn1SEC = 0.027632; // TBD
4   const double rn1SAFE = 0;
5   const double rn1GUAR = 0;
6   const double rn5SEC = 0.2;
7   const double rn5SAFE = 0;
8   const double rn5GUAR = 0;
9   const double rn6SEC = 0.2;
10  const double rn6SAFE = 0;
11  const double rn6GUAR = 0;
12  const double rn7SEC = 0.2;
13  const double rn7SAFE = 0;
14  const double rn7GUAR = 0;

16  formula operational = (n1=0 | n7=0) & (n7=0 | n1=0) ;
```

```
18 module generatedScenario

20 n1: [0..2] init 0;
21 n5: [0..2] init 0;
22 n6: [0..2] init 0;
23 n7: [0..2] init 0;

25 [] (n5=0) & (operational) -> rn5SEC : (n5'=2);
26 [] (n5=2 & n1=0) & (operational) -> rn1SEC : (n1'=2);
27 [] (n5=2 & n6=0) & (operational) -> rn6SEC : (n6'=2);
28 [] (n6=2 & n7=0) & (operational) -> rn7SEC : (n7'=2);

30 endmodule

32 label "systemfailure" = !operational;
33 label "defective" = (n1=1 & n7=1) | (n7=1 & n1=1) ;
34 label "corrupted" = (n1=2 & n7=2) | (n7=2 & n1=2) ;
```

Listing A.2: Example 1 Abstracted Graph

```
1 ctmc

3 const double rn1SEC = 0.2;
4 const double rn1SAFE = 0;
5 const double rn1GUAR = 0;
6 const double rn2SEC = 0.2;
7 const double rn2SAFE = 0;
8 const double rn2GUAR = 0;
9 const double rn3SEC = 0.2;
10 const double rn3SAFE = 0;
11 const double rn3GUAR = 0;
12 const double rn4SEC = 0.2;
13 const double rn4SAFE = 0;
14 const double rn4GUAR = 0;

16 formula operational = (n4=0) ;

18 module generatedScenario

20 n1: [0..2] init 0;
21 n2: [0..2] init 0;
22 n3: [0..2] init 0;
23 n4: [0..2] init 0;

25 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
26 [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
27 [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
28 [] (n3=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
```

```
30  endmodule

32  label "systemfailure" = !operational;
33  label "defective" = (n4=1) ;
34  label "corrupted" = n4=2 ;
```

Listing A.3: Example 1 Module Graph

A.1.2 Example 2: Problematic Security Modularization (Reach-link)

```
1   ctmc

3   const double rn1SEC = 0.15;
4   const double rn1SAFE = 0;
5   const double rn1GUAR = 0;
6   const double rn2SEC = 0.15;
7   const double rn2SAFE = 0;
8   const double rn2GUAR = 0;
9   const double rn3SEC = 0.15;
10  const double rn3SAFE = 0;
11  const double rn3GUAR = 0;
12  const double rn4SEC = 0.15;
13  const double rn4SAFE = 0;
14  const double rn4GUAR = 0;
15  const double rn5SEC = 0.15;
16  const double rn5SAFE = 0;
17  const double rn5GUAR = 0;

19  formula operational = (n1=0 | n5=0) & (n5=0 | n1=0) ;

21  module generatedScenario

23  n1: [0..2] init 0;
24  n2: [0..2] init 0;
25  n3: [0..2] init 0;
26  n4: [0..2] init 0;
27  n5: [0..2] init 0;

29  [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
30  [] (n1=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
31  [] (n2=0) & (operational) -> rn2SEC : (n2'=2);
32  [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
33  [] (n3=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
34  [] (n4=2 & n5=0) & (operational) -> rn5SEC : (n5'=2);

36  endmodule
```

```
38 label "systemfailure" = !operational;
39 label "defective" = (n1=1 & n5=1) | (n5=1 & n1=1) ;
40 label "corrupted" = (n1=2 & n5=2) | (n5=2 & n1=2) ;
```

Listing A.4: Example 2 Original Graph

```
1  ctmc

3  const double rn1SEC = 0.15;
4  const double rn1SAFE = 0;
5  const double rn1GUAR = 0;
6  const double rn3SEC = 0.15;
7  const double rn3SAFE = 0;
8  const double rn3GUAR = 0;
9  const double rn4SEC = 0.15;
10 const double rn4SAFE = 0;
11 const double rn4GUAR = 0;
12 const double rn5SEC = 0.15;
13 const double rn5SAFE = 0;
14 const double rn5GUAR = 0;

16 formula operational = (n1=0 | n5=0) & (n5=0 | n1=0) ;

18 module generatedScenario

20 n1: [0..2] init 0;
21 n3: [0..2] init 0;
22 n4: [0..2] init 0;
23 n5: [0..2] init 0;

25 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
26 [] (n1=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
27 [] (n3=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
28 [] (n4=2 & n5=0) & (operational) -> rn5SEC : (n5'=2);

30 endmodule

32 label "systemfailure" = !operational;
33 label "defective" = (n1=1 & n5=1) | (n5=1 & n1=1) ;
34 label "corrupted" = (n1=2 & n5=2) | (n5=2 & n1=2) ;
```

Listing A.5: Example 2 Abstracted Graph

```
1  ctmc

3  const double rn1SEC = 0.15;
4  const double rn1SAFE = 0;
5  const double rn1GUAR = 0;
6  const double rn2SEC = 0.15;
```

```
7  const double rn2SAFE = 0;
8  const double rn2GUAR = 0;

10 formula operational = (n1=0) ;

12 module generatedScenario

14 n1: [0..2] init 0;
15 n2: [0..2] init 0;

17 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
18 [] (n2=0) & (operational) -> rn2SEC : (n2'=2);

20 endmodule

22 label "systemfailure" = !operational;
23 label "defective" = (n1=1) ;
24 label "corrupted" = n1=2 ;
```

Listing A.6: Example 2 Module Graph

```
1  ctmc

3  const double rn1SEC = 0.15;
4  const double rn1SAFE = 0;
5  const double rn1GUAR = 0;
6  const double rn2SEC = 0.15;
7  const double rn2SAFE = 0;
8  const double rn2GUAR = 0;

10 formula operational = (n1=0) & (n2=0) ;

12 module generatedScenario

14 n1: [0..2] init 0;
15 n2: [0..2] init 0;

17 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
18 [] (n2=0) & (operational) -> rn2SEC : (n2'=2);

20 endmodule

22 label "systemfailure" = !operational;
23 label "defective" = (n1=1) | (n2=1) ;
24 label "corrupted" = n1=2 | n2=2 ;
```

Listing A.7: Example 2 Adapted Module Graph

```
1  ctmc
```

```
 3  const double rn1SEC = 0; // TBD
 4  const double rn1SAFE = 0;
 5  const double rn1GUAR = 0;

 7  formula operational = (n1=0) ;

 9  module generatedScenario

11  n1: [0..2] init 0;

13  endmodule

15  label "systemfailure" = !operational;
16  label "defective" = (n1=1) ;
17  label "corrupted" = n1=2 ;
```

Listing A.8: Example 2 Abstracted for Depth Modularization

```
 1  ctmc

 3  const double rn1SEC = 0.15;
 4  const double rn1SAFE = 0;
 5  const double rn1GUAR = 0;
 6  const double rn2SEC = 0.15;
 7  const double rn2SAFE = 0;
 8  const double rn2GUAR = 0;
 9  const double rn5SEC = 0.06952802547; // TBD
10  const double rn5SAFE = 0;
11  const double rn5GUAR = 0;

13  formula operational = (n1=0 | n5=0) & (n5=0 | n1=0) ;

15  module generatedScenario

17  n1: [0..2] init 0;
18  n2: [0..2] init 0;
19  n5: [0..2] init 0;

21  [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
22  [] (n1=2 & n5=0) & (operational) -> rn5SEC : (n5'=2);
23  [] (n2=0) & (operational) -> rn2SEC : (n2'=2);
24  [] (n2=2 & n5=0) & (operational) -> rn5SEC : (n5'=2);

26  endmodule

28  label "systemfailure" = !operational;
29  label "defective" = (n1=1 & n5=1) | (n5=1 & n1=1) ;
```

```
30 label "corrupted" = (n1=2 & n5=2) | (n5=2 & n1=2) ;
```

Listing A.9: Example 2 Main Graph Depth 1

```
1  ctmc

3  const double rn3SEC = 0.15;
4  const double rn3SAFE = 0;
5  const double rn3GUAR = 0;
6  const double rn4SEC = 0.15;
7  const double rn4SAFE = 0;
8  const double rn4GUAR = 0;
9  const double rn5SEC = 0.15;
10 const double rn5SAFE = 0;
11 const double rn5GUAR = 0;

13 formula operational = (n5=0) ;

15 module generatedScenario

17 n3: [0..2] init 0;
18 n4: [0..2] init 0;
19 n5: [0..2] init 0;

21 [] (n3=0) & (operational) -> rn3SEC : (n3'=2);
22 [] (n3=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
23 [] (n4=2 & n5=0) & (operational) -> rn5SEC : (n5'=2);

25 endmodule

27 label "systemfailure" = !operational;
28 label "defective" = (n5=1) ;
29 label "corrupted" = n5=2 ;
```

Listing A.10: Example 2 Module Graph Depth 2

A.2 Recovery

This section contains the ERIS-generated PRISM code used in the Recovery Chapter 6 and the respective part in the example analysis in Chapter 7, Section 7.2.3.

```
1  ctmc

3  const double rn1SEC = 0.05;
4  const double rn1SAFE = 0.05;
5  const double rn1GUAR = 0;
6  const double rn2SEC = 0.05;
```

```
7  const double rn2SAFE = 0.05;
8  const double rn2GUAR = 0;
9  const double rn3SEC = 0.05;
10 const double rn3SAFE = 0.05;
11 const double rn3GUAR = 0;
12 const double rn4SEC = 0.05;
13 const double rn4SAFE = 0.05;
14 const double rn4GUAR = 0;
15 const double rn5SEC = 0.05;
16 const double rn5SAFE = 0.05;
17 const double rn5GUAR = 0;

19 formula operational = (n3=0 | n4=0) & (n4=0 | n3=0) ;

21 module generatedScenario

23 n1: [0..2] init 0;
24 n2: [0..2] init 0;
25 n3: [0..2] init 0;
26 n4: [0..2] init 0;
27 n5: [0..2] init 0;

29 [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
30 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
31 [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
32 [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
33 [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
34 [] (n2=2) & (operational) -> rn2SAFE : (n2'=1);
35 [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
36 [] (n2=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
37 [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
38 [] (n3=2) & (operational) -> rn3SAFE : (n3'=1);
39 [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
40 [] (n4=2) & (operational) -> rn4SAFE : (n4'=1);
41 [] (n5=0) & (operational) -> rn5SAFE : (n5'=1);

43 endmodule

45 label "systemfailure" = !operational;
46 label "defective" = (n3=1 & n4=1) | (n4=1 & n3=1) ;
47 label "corrupted" = (n3=2 & n4=2) | (n4=2 & n3=2) ;
```

Listing A.11: Without Recovery

```
1  ctmc

3  const double rn1SEC = 0.05;
4  const double rn1SAFE = 0.05;
5  const double rn1GUAR = 0;
```

```
6  const double rn2SEC = 0.05;
7  const double rn2SAFE = 0.05;
8  const double rn2GUAR = 0;
9  const double rn3SEC = 0.05;
10 const double rn3SAFE = 0.05;
11 const double rn3GUAR = 0;
12 const double rn3DEFREC = 0.05;
13 const double rn3CORREC = 0.05;
14 const double rn4SEC = 0.05;
15 const double rn4SAFE = 0.05;
16 const double rn4GUAR = 0;
17 const double rn4DEFREC = 0.05;
18 const double rn4CORREC = 0.05;
19 const double rn5SEC = 0.05;
20 const double rn5SAFE = 0.05;
21 const double rn5GUAR = 0;

23 formula operational = (n3=0 | n4=0) & (n4=0 | n3=0) ;

25 module generatedScenario

27 n1: [0..2] init 0;
28 n2: [0..2] init 0;
29 n3: [0..2] init 0;
30 n4: [0..2] init 0;
31 n5: [0..2] init 0;

33 [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
34 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
35 [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
36 [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
37 [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
38 [] (n2=2) & (operational) -> rn2SAFE : (n2'=1);
39 [] (n2=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
40 [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
41 [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
42 [] (n3=1) & (operational) -> rn3DEFREC : (n3'=0);
43 [] (n3=2) & (operational) -> rn3SAFE : (n3'=1);
44 [] (n3=2) & (operational) -> rn3CORREC : (n3'=0);
45 [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
46 [] (n4=1) & (operational) -> rn4DEFREC : (n4'=0);
47 [] (n4=2) & (operational) -> rn4SAFE : (n4'=1);
48 [] (n4=2) & (operational) -> rn4CORREC : (n4'=0);
49 [] (n5=0) & (operational) -> rn5SAFE : (n5'=1);

51 endmodule

53 label "systemfailure" = !operational;
54 label "defective" = (n3=1 & n4=1) | (n4=1 & n3=1) ;
```

```
55  label "corrupted" = (n3=2 & n4=2) | (n4=2 & n3=2) ;
```

Listing A.12: Strategy **General**: Self-performed

```
1   ctmc

3   const double rn1SEC = 0.05;
4   const double rn1SAFE = 0.05;
5   const double rn1GUAR = 0;
6   const double rn2SEC = 0.05;
7   const double rn2SAFE = 0.05;
8   const double rn2GUAR = 0;
9   const double rn3SEC = 0.05;
10  const double rn3SAFE = 0.05;
11  const double rn3GUAR = 0;
12  const double rn3DEFREC = 0.05;
13  const double rn3CORREC = 0.05;
14  const double rn4SEC = 0.05;
15  const double rn4SAFE = 0.05;
16  const double rn4GUAR = 0;
17  const double rn4DEFREC = 0.05;
18  const double rn4CORREC = 0.05;
19  const double rn5SEC = 0.05;
20  const double rn5SAFE = 0.05;
21  const double rn5GUAR = 0;

23  formula pathesn3DEFREC = n1=0 & n2=0;
24  formula pathesn3CORREC = n1=0 & n2=0;
25  formula pathesn4DEFREC = n1=0 & n2=0;
26  formula pathesn4CORREC = n1=0 & n2=0;
27  formula operational = (n3=0 | n4=0) & (n4=0 | n3=0) ;

29  module generatedScenario

31  n1: [0..2] init 0;
32  n2: [0..2] init 0;
33  n3: [0..2] init 0;
34  n4: [0..2] init 0;
35  n5: [0..2] init 0;

37  [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
38  [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
39  [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
40  [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
41  [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
42  [] (n2=2) & (operational) -> rn2SAFE : (n2'=1);
43  [] (n2=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
44  [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
45  [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
```

```
46  [] (n3=1) & (operational) & (pathesn3DEFREC) -> rn3DEFREC : (n3'=0);
47  [] (n3=2) & (operational) -> rn3SAFE : (n3'=1);
48  [] (n3=2) & (operational) & (pathesn3CORREC) -> rn3CORREC : (n3'=0);
49  [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
50  [] (n4=1) & (operational) & (pathesn4DEFREC) -> rn4DEFREC : (n4'=0);
51  [] (n4=2) & (operational) -> rn4SAFE : (n4'=1);
52  [] (n4=2) & (operational) & (pathesn4CORREC) -> rn4CORREC : (n4'=0);
53  [] (n5=0) & (operational) -> rn5SAFE : (n5'=1);

55  endmodule

57  label "systemfailure" = !operational;
58  label "defective" = (n3=1 & n4=1) | (n4=1 & n3=1) ;
59  label "corrupted" = (n3=2 & n4=2) | (n4=2 & n3=2) ;
```

Listing A.13: Strategy `Restricted`: From outside

```
1   ctmc

3   const double rn1SEC = 0.05;
4   const double rn1SAFE = 0.05;
5   const double rn1GUAR = 0;
6   const double rn2SEC = 0.05;
7   const double rn2SAFE = 0.05;
8   const double rn2GUAR = 0;
9   const double rn3SEC = 0.05;
10  const double rn3SAFE = 0.05;
11  const double rn3GUAR = 0;
12  const double rn3DEFREC = 0.05;
13  const double rn3CORREC = 0.05;
14  const double rn4SEC = 0.05;
15  const double rn4SAFE = 0.05;
16  const double rn4GUAR = 0;
17  const double rn4DEFREC = 0.05;
18  const double rn4CORREC = 0.05;
19  const double rn5SEC = 0.05;
20  const double rn5SAFE = 0.05;
21  const double rn5GUAR = 0;

23  formula pathesn3DEFREC = n5=0;
24  formula pathesn3CORREC = n5=0;
25  formula pathesn4DEFREC = n5=0;
26  formula pathesn4CORREC = n5=0;
27  formula operational = (n3=0 | n4=0) & (n4=0 | n3=0) ;

29  module generatedScenario

31  n1: [0..2] init 0;
32  n2: [0..2] init 0;
```

```
33  n3: [0..2] init 0;
34  n4: [0..2] init 0;
35  n5: [0..2] init 0;

37  [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
38  [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
39  [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
40  [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
41  [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
42  [] (n2=2) & (operational) -> rn2SAFE : (n2'=1);
43  [] (n2=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
44  [] (n2=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
45  [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
46  [] (n3=1) & (operational) & (pathesn3DEFREC) -> rn3DEFREC : (n3'=0);
47  [] (n3=2) & (operational) -> rn3SAFE : (n3'=1);
48  [] (n3=2) & (operational) & (pathesn3CORREC) -> rn3CORREC : (n3'=0);
49  [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
50  [] (n4=1) & (operational) & (pathesn4DEFREC) -> rn4DEFREC : (n4'=0);
51  [] (n4=2) & (operational) -> rn4SAFE : (n4'=1);
52  [] (n4=2) & (operational) & (pathesn4CORREC) -> rn4CORREC : (n4'=0);
53  [] (n5=0) & (operational) -> rn5SAFE : (n5'=1);

55  endmodule

57  label "systemfailure" = !operational;
58  label "defective" = (n3=1 & n4=1) | (n4=1 & n3=1) ;
59  label "corrupted" = (n3=2 & n4=2) | (n4=2 & n3=2) ;
```

Listing A.14: Strategy `Custom`: By another Component

A.3 Automation

This section contains the PRISM code of the autonomous vehicle example of Section 7.4, formerly introduced in Section 4.1.1. Due to the massive code generation of such big systems, only the original system is included in this document (Listing A.15) as it essentially contains the modularizations as well. Disclaimer: ERIS does not perform any optimizations regarding the generation of the operational and essentials formula. For a better readability, they have been shortend by hand (without affecting the semantic).

```
1   ctmc

3   const double rn1SEC = 1.04e-5;
4   const double rn1SAFE = 1.9e-5;
5   const double rn1GUAR = 0;
```

```
6  const double rn2SEC = 1.2e-4;
7  const double rn2SAFE = 1.3e-5;
8  const double rn2GUAR = 0;
9  const double rn3SEC = 1.5e-2;
10 const double rn3SAFE = 1.1e-5;
11 const double rn3GUAR = 0;
12 const double rn4SEC = 1.1e-05;
13 const double rn4SAFE = 1.3e-5;
14 const double rn4GUAR = 0;
15 const double rn5SEC = 1.7e-5;
16 const double rn5SAFE = 1.1e-6;
17 const double rn5GUAR = 0;
18 const double rn6SEC = 0;
19 const double rn6SAFE = 1.01e-6;
20 const double rn6GUAR = 1.9e-6;
21 const double rn6DEFREC = 1.3e-3;
22 const double rn7SEC = 0;
23 const double rn7SAFE = 1.01e-06;
24 const double rn7GUAR = 0;
25 const double rn8SEC = 1.3e-5;
26 const double rn8SAFE = 1.1e-5;
27 const double rn8GUAR = 0;
28 const double rn8DEFREC = 1.5e-5;
29 const double rn9SEC = 1.9e-6;
30 const double rn9SAFE = 1.1e-7;
31 const double rn9GUAR = 0;
32 const double rn10SEC = 1.2e-05;
33 const double rn10SAFE = 1.01e-07;
34 const double rn10GUAR = 0;
35 const double rn11SEC = 1.2e-05;
36 const double rn11SAFE = 1.01e-07;
37 const double rn11GUAR = 0;
38 const double rn12SEC = 1.2e-05;
39 const double rn12SAFE = 1.1e-7;
40 const double rn12GUAR = 0;
41 const double rn13SEC = 1.2e-6;
42 const double rn13SAFE = 1.3e-06;
43 const double rn13GUAR = 0;
44 const double rn14SEC = 1.2e-05;
45 const double rn14SAFE = 1.01e-07;
46 const double rn14GUAR = 0;
47 const double rn15SEC = 1.2e-05;
48 const double rn15SAFE = 1.01e-07;
49 const double rn15GUAR = 0;
50 const double rn16SEC = 1.2e-05;
51 const double rn16SAFE = 1.2e-08;
52 const double rn16GUAR = 0;
53 const double rn17SEC = 0;
54 const double rn17SAFE = 1.3e-06;
```

```
55  const double rn17GUAR = 0;
56  const double rn18SEC = 1.2e-05;
57  const double rn18SAFE = 1.01e-07;
58  const double rn18GUAR = 0;
59  const double rn19SEC = 0;
60  const double rn19SAFE = 1.3e-6;
61  const double rn19GUAR = 0;
62  const double rn20SEC = 1.2e-05;
63  const double rn20SAFE = 1.01e-07;
64  const double rn20GUAR = 0;
65  const double rn21SEC = 1.2e-05;
66  const double rn21SAFE = 1.1e-7;
67  const double rn21GUAR = 0;
68  const double rn22SEC = 1.2e-05;
69  const double rn22SAFE = 1.3e-06;
70  const double rn22GUAR = 0;
71  const double rn23SEC = 1.2e-05;
72  const double rn23SAFE = 1.01e-07;
73  const double rn23GUAR = 0;
74  const double rn24SEC = 0;
75  const double rn24SAFE = 1.3e-06;
76  const double rn24GUAR = 0;
77  const double rn25SEC = 0;
78  const double rn25SAFE = 1.3e-06;
79  const double rn25GUAR = 0;
80  const double rn26SEC = 0;
81  const double rn26SAFE = 1.1e-07;
82  const double rn26GUAR = 0;
83  const double rn27SEC = 0;
84  const double rn27SAFE = 1.1e-07;
85  const double rn27GUAR = 0;
86  const double rn28SEC = 1.2e-05;
87  const double rn28SAFE = 1.3e-06;
88  const double rn28GUAR = 0;
89  const double rn29SEC = 1.1e-06;
90  const double rn29SAFE = 1.02e-6;
91  const double rn29GUAR = 0;
92  const double rn29DEFREC = 0.0011;
93  const double rn29CORREC = 0.0011;
94  const double rn30SEC = 1.1e-07;
95  const double rn30SAFE = 1.02e-07;
96  const double rn30GUAR = 0;
97  const double rn31SEC = 1.1e-06;
98  const double rn31SAFE = 1.02e-06;
99  const double rn31GUAR = 0;
100 const double rn31DEFREC = 0.0011;
101 const double rn31CORREC = 0.0011;
102 const double rn32SEC = 0;
103 const double rn32SAFE = 1.05e-07;
```

```
104 const double rn32GUAR = 0;
105 const double rn33SEC = 0;
106 const double rn33SAFE = 1.01e-06;
107 const double rn33GUAR = 0;
108 const double rn34SEC = 0;
109 const double rn34SAFE = 1.01e-06;
110 const double rn34GUAR = 0;
111 const double rn35SEC = 0;
112 const double rn35SAFE = 1.01e-07;
113 const double rn35GUAR = 0;
114 const double rn36SEC = 0;
115 const double rn36SAFE = 1.01e-06;
116 const double rn36GUAR = 0;
117 const double rn37SEC = 0;
118 const double rn37SAFE = 1.6e-06;
119 const double rn37GUAR = 0;
120 const double rn38SEC = 0;
121 const double rn38SAFE = 1.95e-07;
122 const double rn38GUAR = 0;
123 const double rn39SEC = 0;
124 const double rn39SAFE = 1.6e-06;
125 const double rn39GUAR = 0;
126 const double rn40SEC = 0;
127 const double rn40SAFE = 1.45e-06;
128 const double rn40GUAR = 0;
129 const double rn41SEC = 0;
130 const double rn41SAFE = 1.95e-07;
131 const double rn41GUAR = 0;
132 const double rn42SEC = 0;
133 const double rn42SAFE = 1.05e-07;
134 const double rn42GUAR = 0;
135 const double rn43SEC = 0;
136 const double rn43SAFE = 1.01e-06;
137 const double rn43GUAR = 0;
138 const double rn44SEC = 0;
139 const double rn44SAFE = 1.01e-07;
140 const double rn44GUAR = 0;
141 const double rn45SEC = 0;
142 const double rn45SAFE = 1.01e-06;
143 const double rn45GUAR = 0;
144 const double rn46SEC = 0;
145 const double rn46SAFE = 1.05e-07;
146 const double rn46GUAR = 0;
147 const double rn47SEC = 0;
148 const double rn47SAFE = 1.05e-07;
149 const double rn47GUAR = 0;
150 const double rn48SEC = 0;
151 const double rn48SAFE = 1.01e-07;
152 const double rn48GUAR = 0;
```

```
153 const double rn49SEC = 0;
154 const double rn49SAFE = 1.01e-07;
155 const double rn49GUAR = 0;
156 const double rn50SEC = 0;
157 const double rn50SAFE = 1.3e-6;
158 const double rn50GUAR = 1.3e-6;
159 const double rn51SEC = 1.3890e-5;
160 const double rn51SAFE = 1.1e-6;
161 const double rn51GUAR = 0;
162 const double rn51CORREC = 1.3e-4;

164 formula n2essentials = n7=0;
165 formula n13essentials = n17=0 & n16=0;
166 formula n15essentials = n19=0;
167 formula n21essentials = n20=0;
168 formula n23essentials = n26=0 & n27=0;
169 formula n24essentials = n21=0;
170 formula n25essentials = n21=0;
171 formula n26essentials = n23=0;
172 formula n27essentials = n23=0;
173 formula CUessentials = n39=0 & (n47=0 | n46=0) & (n41=0 | n38=0) & (n49=0 &
         n48=0) & (n45=0 &n44=0) & (n43=0 | n42=0) & (n37=0 | n36=0) & (n35=0 &
         n34=0) & (n33=0 | n32=0) & n7=0;
174 formula operational = (n10=0) & (n11=0) & (n12=0) & (n14=0) & (n15=0 & (n19
        =0)) & (n16=0) & (n18=0) & (n20=0) & (n21=0 & (n20=0)) & (n23=0 & ((n26
        =0 & (n23=0)) & n27=0)) & (n29=0 & CUessentials | n30=0 & CUessentials
        | n31=0 & CUessentials) ;

176 module generatedScenario

178 n1: [0..2] init 0;
179 n2: [0..2] init 0;
180 n3: [0..2] init 0;
181 n4: [0..2] init 0;
182 n5: [0..2] init 0;
183 n6: [0..2] init 0;
184 n7: [0..2] init 0;
185 n8: [0..2] init 0;
186 n9: [0..2] init 0;
187 n10: [0..2] init 0;
188 n11: [0..2] init 0;
189 n12: [0..2] init 0;
190 n13: [0..2] init 0;
191 n14: [0..2] init 0;
192 n15: [0..2] init 0;
193 n16: [0..2] init 0;
194 n17: [0..2] init 0;
195 n18: [0..2] init 0;
196 n19: [0..2] init 0;
```

```
197  n20: [0..2] init 0;
198  n21: [0..2] init 0;
199  n22: [0..2] init 0;
200  n23: [0..2] init 0;
201  n24: [0..2] init 0;
202  n25: [0..2] init 0;
203  n26: [0..2] init 0;
204  n27: [0..2] init 0;
205  n28: [0..2] init 0;
206  n29: [0..2] init 0;
207  n29internalfailure: bool init false;
208  n30: [0..2] init 0;
209  n31: [0..2] init 0;
210  n31internalfailure: bool init false;
211  n32: [0..2] init 0;
212  n33: [0..2] init 0;
213  n34: [0..2] init 0;
214  n35: [0..2] init 0;
215  n36: [0..2] init 0;
216  n37: [0..2] init 0;
217  n38: [0..2] init 0;
218  n39: [0..2] init 0;
219  n40: [0..2] init 0;
220  n41: [0..2] init 0;
221  n42: [0..2] init 0;
222  n43: [0..2] init 0;
223  n44: [0..2] init 0;
224  n45: [0..2] init 0;
225  n46: [0..2] init 0;
226  n47: [0..2] init 0;
227  n48: [0..2] init 0;
228  n49: [0..2] init 0;
229  n50: [0..2] init 0;
230  n51: [0..2] init 0;

232  [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
233  [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
234  [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
235  [] (n1=2 & n8=0) & (n6=0) & (operational) -> rn8SEC-rn6GUAR : (n8'=2);
236  [] (n1=2 & n8=0) & (n6!=0) & (operational) -> rn8SEC : (n8'=2);
237  [] (n2=0) & (!n2essentials) & (operational)-> (n2'=1);
238  [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
239  [] (n2=2) & (operational) -> rn2SAFE : (n2'=1);
240  [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
241  [] (n3=2) & (operational) -> rn3SAFE : (n3'=1);
242  [] (n3=2 & n8=0) & (n6=0) & (operational) -> rn8SEC-rn6GUAR : (n8'=2);
243  [] (n3=2 & n8=0) & (n6!=0) & (operational) -> rn8SEC : (n8'=2);
244  [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
245  [] (n4=0) & (operational) -> rn4SEC : (n4'=2);
```

```
246  [] (n4=2) & (operational) -> rn4SAFE : (n4'=1);
247  [] (n4=2 & n8=0) & (n6=0) & (operational) -> rn8SEC-rn6GUAR : (n8'=2);
248  [] (n4=2 & n8=0) & (n6!=0) & (operational) -> rn8SEC : (n8'=2);
249  [] (n5=0) & (operational) -> rn5SAFE : (n5'=1);
250  [] (n5=0) & (n6=0) & (operational) -> rn5SEC-rn6GUAR : (n5'=2);
251  [] (n5=0) & (n6!=0) & (operational) -> rn5SEC : (n5'=2);
252  [] (n5=2) & (operational) -> rn5SAFE : (n5'=1);
253  [] (n5=2 & n9=0) & (n50=0) & (operational) -> rn9SEC-rn50GUAR : (n9'=2);
254  [] (n5=2 & n9=0) & (n50!=0) & (operational) -> rn9SEC : (n9'=2);
255  [] (n6=0) & (operational) -> rn6SAFE : (n6'=1);
256  [] (n6=1) & (operational) -> rn6DEFREC : (n6'=0);
257  [] (n7=0) & (operational) -> rn7SAFE : (n7'=1);
258  [] (n8=0) & (operational) -> rn8SAFE : (n8'=1);
259  [] (n8=1) & (operational) -> rn8DEFREC : (n8'=0);
260  [] (n8=2) & (operational) -> rn8SAFE : (n8'=1);
261  [] (n8=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
262  [] (n8=2 & n9=0) & (n50=0) & (operational) -> rn9SEC-rn50GUAR : (n9'=2);
263  [] (n8=2 & n9=0) & (n50!=0) & (operational) -> rn9SEC : (n9'=2);
264  [] (n8=2 & n1=0) & (operational) -> rn1SEC : (n1'=2);
265  [] (n8=2 & n4=0) & (operational) -> rn4SEC : (n4'=2);
266  [] (n8=2 & n3=0) & (operational) -> rn3SEC : (n3'=2);
267  [] (n9=0) & (operational) -> rn9SAFE : (n9'=1);
268  [] (n9=2) & (operational) -> rn9SAFE : (n9'=1);
269  [] (n9=2 & n18=0) & (operational) -> rn18SEC : (n18'=2);
270  [] (n9=2 & n28=0) & (operational) -> rn28SEC : (n28'=2);
271  [] (n9=2 & n31=0) & (operational) -> rn31SEC : (n31'=2);
272  [] (n9=2 & n30=0) & (operational) -> rn30SEC : (n30'=2);
273  [] (n9=2 & n29=0) & (operational) -> rn29SEC : (n29'=2);
274  [] (n9=2 & n12=0) & (operational) -> rn12SEC : (n12'=2);
275  [] (n9=2 & n16=0) & (operational) -> rn16SEC : (n16'=2);
276  [] (n9=2 & n8=0) & (n6=0) & (operational) -> rn8SEC-rn6GUAR : (n8'=2);
277  [] (n9=2 & n8=0) & (n6!=0) & (operational) -> rn8SEC : (n8'=2);
278  [] (n9=2 & n15=0) & (operational) -> rn15SEC : (n15'=2);
279  [] (n9=2 & n14=0) & (operational) -> rn14SEC : (n14'=2);
280  [] (n9=2 & n23=0) & (operational) -> rn23SEC : (n23'=2);
281  [] (n9=2 & n22=0) & (operational) -> rn22SEC : (n22'=2);
282  [] (n9=2 & n21=0) & (operational) -> rn21SEC : (n21'=2);
283  [] (n9=2 & n20=0) & (operational) -> rn20SEC : (n20'=2);
284  [] (n9=2 & n10=0) & (operational) -> rn10SEC : (n10'=2);
285  [] (n9=2 & n13=0) & (operational) -> rn13SEC : (n13'=2);
286  [] (n9=2 & n11=0) & (operational) -> rn11SEC : (n11'=2);
287  [] (n10=0) & (operational) -> rn10SAFE : (n10'=1);
288  [] (n10=2) & (operational) -> rn10SAFE : (n10'=1);
289  [] (n11=0) & (operational) -> rn11SAFE : (n11'=1);
290  [] (n11=2) & (operational) -> rn11SAFE : (n11'=1);
291  [] (n12=0) & (operational) -> rn12SAFE : (n12'=1);
292  [] (n12=2) & (operational) -> rn12SAFE : (n12'=1);
293  [] (n13=0) & (!n13essentials) & (operational)-> (n13'=1);
294  [] (n13=0) & (operational) -> rn13SAFE : (n13'=1);
```

```
295  [] (n13=2) & (operational) -> rn13SAFE : (n13'=1);
296  [] (n14=0) & (operational) -> rn14SAFE : (n14'=1);
297  [] (n14=2) & (operational) -> rn14SAFE : (n14'=1);
298  [] (n15=0) & (!n15essentials) & (operational)-> (n15'=1);
299  [] (n15=0) & (operational) -> rn15SAFE : (n15'=1);
300  [] (n15=2) & (operational) -> rn15SAFE : (n15'=1);
301  [] (n16=0) & (operational) -> rn16SAFE : (n16'=1);
302  [] (n16=2) & (operational) -> rn16SAFE : (n16'=1);
303  [] (n17=0) & (operational) -> rn17SAFE : (n17'=1);
304  [] (n18=0) & (operational) -> rn18SAFE : (n18'=1);
305  [] (n18=2) & (operational) -> rn18SAFE : (n18'=1);
306  [] (n19=0) & (operational) -> rn19SAFE : (n19'=1);
307  [] (n20=0) & (operational) -> rn20SAFE : (n20'=1);
308  [] (n20=2) & (operational) -> rn20SAFE : (n20'=1);
309  [] (n21=0) & (!n21essentials) & (operational)-> (n21'=1);
310  [] (n21=0) & (operational) -> rn21SAFE : (n21'=1);
311  [] (n21=2) & (operational) -> rn21SAFE : (n21'=1);
312  [] (n22=0) & (operational) -> rn22SAFE : (n22'=1);
313  [] (n22=2) & (operational) -> rn22SAFE : (n22'=1);
314  [] (n23=0) & (!n23essentials) & (operational)-> (n23'=1);
315  [] (n23=0) & (operational) -> rn23SAFE : (n23'=1);
316  [] (n23=2) & (operational) -> rn23SAFE : (n23'=1);
317  [] (n24=0) & (!n24essentials) & (operational)-> (n24'=1);
318  [] (n24=0) & (operational) -> rn24SAFE : (n24'=1);
319  [] (n25=0) & (!n25essentials) & (operational)-> (n25'=1);
320  [] (n25=0) & (operational) -> rn25SAFE : (n25'=1);
321  [] (n26=0) & (!n26essentials) & (operational)-> (n26'=1);
322  [] (n26=0) & (operational) -> rn26SAFE : (n26'=1);
323  [] (n27=0) & (!n27essentials) & (operational)-> (n27'=1);
324  [] (n27=0) & (operational) -> rn27SAFE : (n27'=1);
325  [] (n28=0) & (operational) -> rn28SAFE : (n28'=1);
326  [] (n28=2) & (operational) -> rn28SAFE : (n28'=1);
327  [] (n29=0) & (!CUessentials) & (operational)-> (n29'=1);
328  [] (n29=0) & (operational) -> rn29SAFE : (n29'=1) & (n29internalfailure'=
      true);
329  [] (n29=1) & (operational) & (n29internalfailure)-> rn29DEFREC : (n29'=0) &
       (n29internalfailure'=false);
330  [] (n29=2) & (operational) -> rn29SAFE : (n29'=1);
331  [] (n29=2 & n9=0) & (n50=0) & (operational) -> rn9SEC-rn50GUAR : (n9'=2);
332  [] (n29=2 & n9=0) & (n50!=0) & (operational) -> rn9SEC : (n9'=2);
333  [] (n29=2 & n51=0) & (n6=0) & (operational) -> rn51SEC-rn6GUAR : (n51'=2);
334  [] (n29=2 & n51=0) & (n6!=0) & (operational) -> rn51SEC : (n51'=2);
335  [] (n29=2) & (operational) -> rn29CORREC : (n29'=0);
336  [] (n30=0) & (!CUessentials) & (operational)-> (n30'=1);
337  [] (n30=0) & (operational) -> rn30SAFE : (n30'=1);
338  [] (n30=2) & (operational) -> rn30SAFE : (n30'=1);
339  [] (n30=2 & n9=0) & (n50=0) & (operational) -> rn9SEC-rn50GUAR : (n9'=2);
340  [] (n30=2 & n9=0) & (n50!=0) & (operational) -> rn9SEC : (n9'=2);
341  [] (n30=2 & n51=0) & (n6=0) & (operational) -> rn51SEC-rn6GUAR : (n51'=2);
```

```
342  [] (n30=2 & n51=0) & (n6!=0) & (operational) -> rn51SEC : (n51'=2);
343  [] (n31=0) & (!CUessentials) & (operational)-> (n31'=1);
344  [] (n31=0) & (operational) -> rn31SAFE : (n31'=1) & (n31internalfailure'=
       true);
345  [] (n31=1) & (operational) & (n31internalfailure)-> rn31DEFREC : (n31'=0) &
        (n31internalfailure'=false);
346  [] (n31=2) & (operational) -> rn31SAFE : (n31'=1);
347  [] (n31=2 & n9=0) & (n50=0) & (operational) -> rn9SEC-rn50GUAR : (n9'=2);
348  [] (n31=2 & n9=0) & (n50!=0) & (operational) -> rn9SEC : (n9'=2);
349  [] (n31=2 & n51=0) & (n6=0) & (operational) -> rn51SEC-rn6GUAR : (n51'=2);
350  [] (n31=2 & n51=0) & (n6!=0) & (operational) -> rn51SEC : (n51'=2);
351  [] (n31=2) & (operational) -> rn31CORREC : (n31'=0);
352  [] (n32=0) & (operational) -> rn32SAFE : (n32'=1);
353  [] (n33=0) & (operational) -> rn33SAFE : (n33'=1);
354  [] (n34=0) & (operational) -> rn34SAFE : (n34'=1);
355  [] (n35=0) & (operational) -> rn35SAFE : (n35'=1);
356  [] (n36=0) & (operational) -> rn36SAFE : (n36'=1);
357  [] (n37=0) & (operational) -> rn37SAFE : (n37'=1);
358  [] (n38=0) & (operational) -> rn38SAFE : (n38'=1);
359  [] (n39=0) & (operational) -> rn39SAFE : (n39'=1);
360  [] (n40=0) & (operational) -> rn40SAFE : (n40'=1);
361  [] (n41=0) & (operational) -> rn41SAFE : (n41'=1);
362  [] (n42=0) & (operational) -> rn42SAFE : (n42'=1);
363  [] (n43=0) & (operational) -> rn43SAFE : (n43'=1);
364  [] (n44=0) & (operational) -> rn44SAFE : (n44'=1);
365  [] (n45=0) & (operational) -> rn45SAFE : (n45'=1);
366  [] (n46=0) & (operational) -> rn46SAFE : (n46'=1);
367  [] (n47=0) & (operational) -> rn47SAFE : (n47'=1);
368  [] (n48=0) & (operational) -> rn48SAFE : (n48'=1);
369  [] (n49=0) & (operational) -> rn49SAFE : (n49'=1);
370  [] (n50=0) & (operational) -> rn50SAFE : (n50'=1);
371  [] (n51=0) & (operational) -> rn51SAFE : (n51'=1);
372  [] (n51=0) & (n6=0) & (operational) -> rn51SEC-rn6GUAR : (n51'=2);
373  [] (n51=0) & (n6!=0) & (operational) -> rn51SEC : (n51'=2);
374  [] (n51=2) & (operational) -> rn51SAFE : (n51'=1);
375  [] (n51=2 & n31=0) & (operational) -> rn31SEC : (n31'=2);
376  [] (n51=2 & n30=0) & (operational) -> rn30SEC : (n30'=2);
377  [] (n51=2 & n29=0) & (operational) -> rn29SEC : (n29'=2);
378  [] (n51=2) & (operational) -> rn51CORREC : (n51'=0);

380  endmodule

382  label "systemfailure" = !operational;
383  label "defective" = (n10=1) | (n11=1) | (n12=1) | (n14=1) | (n15=1 | (n19
       =1)) | (n16=1) | (n18=1) | (n20=1) | (n21=1 | (n20=1)) | (n23=1 | ((n26
       =1 | (n23=1)) | n27=1)) | (n29=1 | (n39=1 | (n47=1 & n46=1) | (n41=1 &
       n38=1) | (n49=1 | n48=1) | (n45=1 |n44=1) | (n43=1 & n42=1) | (n37=1 &
       n36=1) | (n35=1 | n34=1) | (n33=1 & n32=1) | n7=1) & n30=1 | (n39=1 | (
       n47=1 & n46=1) | (n41=1 & n38=1) | (n49=1 | n48=1) | (n45=1 |n44=1) | (
```

```
      n43=1 & n42=1) | (n37=1 & n36=1) | (n35=1 | n34=1) | (n33=1 & n32=1) |
      n7=1) & n31=1 | (n39=1 | (n47=1 & n46=1) | (n41=1 & n38=1) | (n49=1 |
      n48=1) | (n45=1 |n44=1) | (n43=1 & n42=1) | (n37=1 & n36=1) | (n35=1 |
      n34=1) | (n33=1 & n32=1) | n7=1));
384 label "corrupted" = n10=2 | n11=2 | n12=2 | n14=2 | n15=2 | n16=2 | n18=2 |
       n20=2 | n21=2 | n23=2 | (n29=2 & n30=2 & n31=2) | (n30=2 & n29=2 & n31
      =2) | (n31=2 & n30=2 & n29=2) ;
```

Listing A.15: Original: From outside

A.4 Related Work

This section contains the ERIS-generated PRISM code used in the Related Work Chapter 8.

```
1  ctmc

3  const double rn1SEC = 0.20;
4  const double rn1SAFE = 0.1;
5  const double rn1GUAR = 0;
6  const double rn2SEC = 0.05;
7  const double rn2SAFE = 0.2;
8  const double rn2GUAR = 0;
9  const double rn3SEC = 0;
10 const double rn3SAFE = 0.3;
11 const double rn3GUAR = 0.20;
12 const double rn4SEC = 0.25;
13 const double rn4SAFE = 0.15;
14 const double rn4GUAR = 0;

16 formula n4essentials = n1=0 | n2=0 ;
17 formula operational = (n4=0 & (n1=0 | n2=0 )) ;

19 module generatedScenario

21 n1: [0..2] init 0;
22 n2: [0..2] init 0;
23 n3: [0..2] init 0;
24 n4: [0..2] init 0;

26 [] (n1=0) & (operational) -> rn1SAFE : (n1'=1);
27 [] (n1=0) & (operational) -> rn1SEC : (n1'=2);
28 [] (n1=2) & (operational) -> rn1SAFE : (n1'=1);
29 [] (n1=2 & n2=0) & (operational) -> rn2SEC : (n2'=2);
30 [] (n2=0) & (operational) -> rn2SAFE : (n2'=1);
31 [] (n2=0) & (operational) -> rn2SEC : (n2'=2);
```

```
32 [] (n2=2) & (operational) -> rn2SAFE : (n2'=1);
33 [] (n2=2 & n4=0) & (n3=0) & (operational) -> rn4SEC-rn3GUAR : (n4'=2);
34 [] (n2=2 & n4=0) & (n3!=0) & (operational) -> rn4SEC : (n4'=2);
35 [] (n3=0) & (operational) -> rn3SAFE : (n3'=1);
36 [] (n4=0) & (!n4essentials) & (operational)-> (n4'=1);
37 [] (n4=0) & (operational) -> rn4SAFE : (n4'=1);
38 [] (n4=2) & (operational) -> rn4SAFE : (n4'=1);

40 endmodule

42 label "systemfailure" = !operational;
43 label "defective" = (n4=1 | (n1=1 & n2=1 )) ;
44 label "corrupted" = n4=2 ;
```

Listing A.16: AT-CARS and ERIS Comparison Model

B Evaluation Results

This chapter contains the results of all performed PRISM and AT-CARS computations needed to generate the plots and graphics of this thesis.

B.1 Modularization

Subsequent tables contain specific results of the example evaluations of Chapter 5, more precisely Section 5.3.

Table B.1: Example 1: Result Probabilities and Module Rates at Time T

T	Probability in Percent (rounded)			
	Original	Modularization	Original	Modularization
	attack_rate(n_5) = 0.2		attack_rate(n_5) = 1	
0	0.0	0.0	0.0	0.0
1	0000029	0.0000049	0.000013	0.000021
2	0.00028	0.00047	0.0012	0.0018
3	0.0036	0.0059	0.0138	0.0197
4	0.0207	0.0328	0.0723	0.0987
5	0.0753	0.1166	0.2439	0.3194
6	0.2068	0.3129	0.6224	0.7869
7	0.4675	0.6919	0.0131	1.6109
8	0.9176	1.3291	2.4124	2.8855
9	1.6178	2.2954	3.9963	4.6755
10	2.6228	3.6480	6.1071	7.0096
11	3.9753	5.4248	8.7525	9.8806
12	5.7028	7.6414	11.9078	13.2496
13	7.8159	10.2911	15.5211	17.0534
14	10.3081	13.3476	19.5212	21.2127
15	13.1582	16.7679	23.825	25.6397
16	16.3323	20.4976	28.3447	30.2448
17	19.7868	24.4743	32.9935	34.9419
18	23.4718	28.6325	37.69	39.6523
19	27.3341	32.9065	42.3612	44.3066
20	31.3194	37.2338	46.9441	48.8466
21	35.3751	41.5568	51.387	53.2251
22	39.4516	45.8242	55.6486	57.4056
23	43.5036	49.9918	59.6981	61.3617
24	47.4912	54.0232	63.5141	65.0759
25	51.3799	57.8892	67.0831	68.5383
26	55.1411	61.5676	70.3989	71.7455
27	58.7517	65.0426	73.4608	74.6992
28	62.1941	68.3044	76.2729	77.4055
29	65.4555	71.3479	78.8428	79.8734
30	68.5277	74.1724	81.1808	82.1142
31	71.4063	76.7806	83.2991	84.1409
32	74.0902	79.178	85.2112	85.9674
33	76.5812	81.3725	86.9311	87.608
34	78.8833	83.3732	88.4732	89.0772
35	81.0026	85.1908	89.8519	90.3891
36	82.9464	86.8365	91.0811	91.5576
37	84.723	88.3218	92.1743	92.5957
38	86.3418	89.6584	93.1441	93.516
39	87.8121	90.858	94.0026	94.33
40	89.1439	91.9318	94.761	95.0486

Table B.2: Example 2: Result Probabilities and Module Rates at Time T

T	Probability in Percent (rounded)			
	Original	Simple Modularization	Depth-2 Modularization	Adapted Modularization
	attack_rate(n_i) = 0.15			
0	0.0	0.0	0.0	0.0
1	0.0021	0.0019	0.0041	0.0036
2	0.0330	0.0266	0.0617	0.0502
3	0.1589	0.1195	0.2887	0.2194
4	0.4686	0.3358	0.8316	0.5999
5	1.0543	0.7292	1.8311	1.2691
6	1.9955	1.3459	3.3967	2.2838
7	3.3494	2.2208	5.5932	3.6775
8	5.1472	3.3769	8.4376	5.461
9	7.3941	4.8248	11.9027	7.626
10	10.0721	6.5642	15.9252	10.1484
11	13.1444	8.5854	20.4158	12.9927
12	16.5605	10.8708	25.2686	16.1158
13	20.2613	13.3969	30.3716	19.4697
14	24.1828	16.1357	35.6133	23.005
15	28.2607	19.0567	40.8894	26.6727
16	32.4328	22.1277	46.1066	30.4258
17	36.6411	25.3163	51.1854	34.2207
18	40.8334	28.5908	56.0608	38.0177
19	44.9642	31.9205	60.6829	41.7816
20	48.9948	35.2768	65.0156	45.4821
21	52.8935	38.6333	69.0358	49.0931
22	56.6353	41.9662	72.7314	52.5936
23	60.2013	45.2541	76.0998	55.9664
24	63.5784	48.47	79.1461	59.1983

B.2 Recovery

The following Table B.3 contains the detailed evaluation results of the analysed example of Chapter 6.

Table B.3: Result Probabilities and Input Rates at Time T

T	Probability in Percent (rounded)			
	Without Recovery	Self-performed Recovery	Recovery from the outside	Recovery by another component
0	0.0	0.0	0.0	0.0
1	0.2381	0.2343	0.2345	0.2343
2	0.9085	0.8804	0.8843	0.8818
3	1.9525	1.8655	1.8827	1.8715
4	3.3183	3.1279	3.1766	3.1458
5	4.9593	4.6168	4.7219	4.6563
6	6.8330	6.2877	6.4803	6.3620
7	8.9006	8.1027	8.4182	8.2275
8	11.1266	10.0291	10.5053	10.2223
9	13.4788	12.0386	12.7144	12.3197
10	15.9281	14.1074	15.0205	14.4965
11	18.4484	16.2149	17.4012	16.7323
12	21.0166	18.3437	19.8363	19.0097
13	23.6123	20.4793	22.3076	21.3131
14	26.2177	22.6096	24.7989	23.6294
15	28.8175	24.7245	27.2960	25.9470
16	31.3984	26.8159	29.7862	28.2560
17	33.9493	28.8772	32.2588	30.5481
18	36.4610	30.9031	34.7044	32.8161
19	38.9258	32.8895	37.1150	35.0539
20	41.3373	34.8333	39.4839	37.2565
21	43.6908	36.7323	41.8055	39.4197
22	45.9822	38.5849	44.0753	41.5401
23	48.2088	40.3900	46.2897	43.6149
24	50.3685	42.1047	48.4457	45.6420

B.3 Automation Analysis Results

The subsequent tables show the results computed for analysis of the system example in Chapter 7, Section 7.4. Table B.6 shows one simulation run of AT-CARS as an example.

Table B.4: AT-CARS Module Rates

Month	Probability Rate	
	Failure (defective)	Attack (corrupted)
0	0.0	0.0
1	0.00109745	0.0
2	0.00103438	9.70259e-06
3	0.00105395	9.53744e-06
4	0.00108267	1.71487e-05
5	0.001108	3.39089e-05
6	0.00113479	5.12639e-05
7	0.00116753	6.31768e-05
8	0.00120027	7.47584e-05
9	0.00123662	8.20148e-05
10	0.00127123	8.99793e-05
11	0.00130127	0.000101265
12	0.00133847	0.000104189
13	0.00137536	0.000106061
14	0.00140833	0.000110367
15	0.00144031	0.000114167
16	0.00146973	0.000118984
17	0.00150055	0.000120901
18	0.00153321	0.000119487
19	0.0015617	0.000120577
20	0.00158549	0.000124665
21	0.00160904	0.000127389
22	0.00163189	0.000129182
23	0.0016516	0.0001324
24	0.00167	0.00013524
Computation Time	336 minutes	

Table B.5: Option 3 Evaluation with AT-CARS Module

Month	Probability in Percent
	System Failure
0	0.0
1	0.354747
2	0.708293
3	1.06649
4	1.43109
5	1.79881
6	2.17149
7	2.55242
8	2.93878
9	3.33342
10	3.7324
11	4.13249
12	4.54467
13	4.96216
14	5.38046
15	5.80226
16	6.22512
17	6.65385
18	7.08921
19	7.52228
20	7.95126
21	8.38242
22	8.81475
23	9.2437
24	9.67186
Computation Time	229 minutes

Table B.6: AT-CARS Example Iteration

Event	System Time in h	Failed Element	Event Description
0	0.0		Mission Start
1	129.8407	SW Mission Planning	Failure + Isolation Start
2	129.8407	SW Mission Planning	Isolated + Waiting for Reconfiguration
3	280.3914	SW Detection	Failure + Isolation Start
4	280.3914	SW Detection	Isolated + Switching Start
5	280.3914	SW Detection	Switched + Waiting for Reconfiguration
6	740.9053	SW Mission Planning	Failure + Isolation Start
7	740.9053	SW Mission Planning	Isolated + Switching Start
8	740.9053	SW Mission Planning	Switched + Waiting for Reconfiguration
9	4718.2199	SW Prediction	Failure + Isolation Start
10	4718.2199	SW Prediction	Isolated + Switching Start
11	4718.2199	SW Prediction	Switched + Waiting for Reconfiguration
12	5231.1182	HDL n_{39}	Failure
13	5231.1182	SW_E Detection	Failure + Isolation Start
14	5231.1182	SW_E Detection	Isolated + Waiting for Reconfiguration
15	6838.3127	Localization	Failure + Isolation Start
16	6838.3127	Localization	Isolated + Waiting for Reconfiguration
17	7419.0539	$CU2$ n_{31}	Failure
18	7419.0539	SW Detection	Failure + Isolation Start
19	7419.0539	SW Prediction	Failure + Isolation Start
20	7419.0539	SW Motion Planning	Failure + Isolation Start
21	7419.0539	SW Actuation	Failure + Isolation Start
22	7419.0539	SW Detection	Isolated + Switching failed
23	17520		Mission Failed

Note that SW refers to the software application (defined by Autoware [Kat+18], see also Section 2.2.3) running on the CUs as active and active hot instances. SW_E is the respective degraded version running on the emergency computing unit CUE.

Table B.7: Corruption Result Probabilities

Month	Probability in Percent		
	Corruption of n_5 (OBD-II interface)	Corruption of n_8 (Infotainment)	Corruption of n_{51} (VANET Router)
0	0.0		
1	1.09346	6.41461e-03	0.829384
2	2.16941	0.0250678	1.57323
3	3.22798	0.0551039	2.24022
4	4.26931	0.095706	2.83813
5	5.29364	0.146099	3.37406
6	6.30111	0.205542	3.85431
7	7.29177	0.273326	4.28448
8	8.26587	0.348783	4.66974
9	9.22342	0.431268	5.01459
10	10.1648	0.520185	5.32326
11	11.0903	0.614975	5.59957
12	11.9995	0.715048	5.84646
13	12.8928	0.81991	6.06713
14	13.7709	0.929102	6.26444
15	14.6337	1.04215	6.44069
16	15.4816	1.15864	6.59815
17	16.3143	1.27813	6.73853
18	17.132	1.40021	6.86351
19	17.9356	1.52461	6.97507
20	18.7255	1.65102	7.07472
21	19.5015	1.77906	7.16339
22	20.2637	1.90844	7.24224
23	21.0129	2.03895	7.31252
24	21.7489	2.1703	7.37497
Computation Time	248 minutes	228 minutes	232 minutes