Engineering of Safety in Automated Safety-Critical Systems through Design-time Verification and Runtime Validation of Environment Assumptions

Doctoral Thesis
(Dissertation)

to be awarded the degree of
Doctor rerum naturalium
(Dr. rer. nat.)

submitted by

Adina Aniculăesei
from Iaşi, Iaşi County, Romania

approved by the Faculty of Mathematics, Computer Science
and Mechanical Engineering,
Clausthal University of Technology

2022

Dissertation Clausthal, SSE-Dissertation XXX, 2022

Chairperson of the Board of Examiners
Prof. Dr. Der Die Was

Chief Reviewer
Prof. Dr. Andreas Rausch

2. Reviewer
Prof. Dr. Stefan Wagner

3. Reviewer
Prof. Dr.-Ing. habil. Alois Christian Knoll

Date of oral examination: XX.XX.XXXX

Abstract

Automated systems are often used in highly safety- and mission-critical applications. Failure of automated systems could lead to mission failure and endanger human life. Being complex and safety- and mission-critical in their nature, automated safety-critical systems benefit from a structured and rigorous system development process with clearly specified tasks for verification and validation in order to make sure that such systems are safe. Formal verification methods used at design-time can provide proof of the system's correctness with respect to a safety property specification. Yet, design-time verification methods can reason only on the basis of the information which is available to the systems engineers at design-time. Furthermore, design-time verification techniques have scalability issues, which limits the size of the systems that can be verified. Testing is used to complement design-time verification and overcome its scalability issues. During system test, test oracles in form of property monitors check whether the behavior of the system fulfills the system requirements.

During the system operation, unpredicted events occurring in the operational environment result in safety hazards. These hazards do not appear as consequence of system faults, but because assumptions made about the operational environment during design-time are not valid anymore. The property monitors designed to check the system requirements cannot detect the assumptions violation, because there is no explicit definition of the environment assumptions at system's design-time.

In order to address these limitations, this thesis proposes an engineering approach which extends the quality assurance goals for automated safety-critical systems towards the verification and validation of environment assumptions. During system design, environment assumptions are explicitly defined and monitors are derived from the environment assumptions definition. During system testing, the quality assurance goals are to test the environment assumptions monitor as well as the implemented system. Requirements validation means both the validation of the system safety requirements as well as the validation of the environment assumptions. The approach is integrated in the system development process and is evaluated on the basis of two cases studies: a mobile service robot and an automotive function for accurate vehicle speed estimation.

Acknowledgment

This thesis is the result of many years of research at the Institute of Software and Systems Engineering (ISSE) of TU Clausthal and would not have been possible without the support many people throughout this time. I would like to thank my doctoral supervisor Prof. Dr. Andreas Rausch for his guidance throughout the development and writing of this thesis. I especially want to thank Prof. Dr. Andreas Rausch for his patience with me in the first meetings in which we discussed my PhD topic. He managed to create an environment for the fruitful exchange of ideas through lively discussions on the topic of this thesis as well as exciting meetings in various research projects carried throughout my years as a doctoral researcher. I feel that the talks we had in these meetings have enriched me and have contributed significantly to my professional and personal development. I would like to also thank Prof. Dr. Stefan Wagner and Prof. Dr. habil. Alois Knoll for taking over the task to revise this thesis as external reviewers.

I would further like to thank several past and present colleagues whose suggestions and feedback have been a valuable input for my work on this thesis. I thank my former colleagues, Prof. Dr. Sebastian Herold, Dr. Constanze Deiters, Dr. Christian Bartelt, Dr. Thomas Schäfer (née Ternite), Dr. Christian Bartelt, Dr. Dirk Niebuhr, and Dr. Holger Klus. They made me feel welcome and helped me adjust in my first years at the Chair of Sofware Systems Engineering, which is the precursor of ISSE. Further thanks go to Dr. Malte Mauritz and Dr. Stefan Ruhl. Their PhD theses have served as an inspiration on how a PhD thesis should be written. I would like to further thank former colleagues Tim Warnecke, Jörg Grieser and Karina Rehfeldt. I am grateful for the fruitful discussions we had during the projects we worked on together on the *iServeU* and *VanAssist* projects. I would also like to thank Daniel Arnsberger, Peer Denecke, Jan Toennemann, and Andreas Vorwald, with whom I had the opportunity of writing exciting papers on the verification and validation of autonomous robots and automated vehicle functions, some of which are featured in this thesis.

Special thanks go to Meng Zhang and Andreas Vorwald for the great collaboration and the fun we had in the projects *Accurate Speed Estimation of Verification of Ego-Vehicle Speed* and *Verification of Exhaust Aftertreatment Systems*. Meng Zhang and Andreas Vorwald receive further thanks for taking over many of the administrative tasks in the research group of Dependable and Autonomous Cyber-Physical Systems during the final months of writing this thesis.

I would like to thank Dr. Christoph Knieke, Dr. Marco Körner, and Henrik Lisner (née Peters) for providing valuable support during the projects carried out in collaboration with Volkswagen. I would also like to thank Prof. Dr. Marco Kuhrmann, Dr. Christoph Knieke and Dr. Peter Engel for the exciting time we had during our teaching activities in the Software Systems Engineering and Informatik I lectures.

I want to thank Steffen Küpper, Sebastian Lawrenz, Dirk Herrling and Mirco Schindler for helping me navigate the working environment at ISSE and providing valuable input on how to find my feet on the sail ship Petrine, even though I do not know how to swim. I would also like to thank Thomas Bravin, Simone Dahms, and Nadine Hahn for their support with any technical and not so technical questions I had during the years, getting me out of trouble more times than I can count, and lending a listening ear during the coffee breaks.

I want to thank Katharina and Holger Peters, and their son Erik for their friendship and the home cinema weekends they organized throughout the years, which were a great outlet for laughter and relaxation.

My deepest thank you goes to my parents, Gherghina and Vasile, for their unwavering encouragement and support throughout my school and university education. This thesis is dedicated to you.

Contents

Abstract i

Contents iv

List of Figures vii

List of Tables x

1. Introduction 3
1.1. Motivation 3
1.2. Goals and Contributions of this Thesis 5
1.3. Thesis Structure 6

2. Fundamental Concepts and Approaches 9
2.1. Definitions of the Problem Domain 10
2.1.1. Safety-critical Systems 10
2.1.2. Functional Safety and Safety of the Intended Functionality 11
2.1.3. Automated Systems versus Autonomous Systems 13
2.1.4. Uncertain Environments 18
2.2. System Life Cycle and Development Process 19
2.2.1. System Development Process according to ISO 26262 20
2.2.2. System Development Process according to ISO/PAS 21448 29
2.3. Property Specification for Automated Safety-critical Systems 36
2.3.1. Timed Computation Tree Logic 37
2.3.2. Probabilistic Computational Tree Logic 40
2.4. System Modeling for Automated Safety-Critical Systems 44
2.4.1. Timed Automata 45
2.4.2. Short Overview of Modeling Formalisms for Probabilistic Systems 47
2.4.3. Markov Decision Processes 49
2.5. Verification and Validation of System Properties in Automated Safety-critical Systems 53
2.5.1. Testing 54
2.5.2. Design-time Verification 58
2.5.3. Runtime Verification 69
2.6. Summary 74

3. Problem Analysis 75

3.1. Motivational Example: Mobile Service Robot 76
3.2. Overall Development Process 81
3.3. Requirements Elicitation and Analysis 83
3.3.1. Informal Specification of System Requirements 83
3.3.2. Formal Specification of System Requirements 88
3.4. Safety Analysis 95
3.5. System Design 100
3.5.1. Usage of Formal Models in System Design 101
3.5.2. Environment Model 102
3.5.3. Technical System Model 104
3.5.4. Overall System Model 104
3.5.5. Specification of System Properties 108
3.5.6. Design-Time Verification 109
3.6. System Implementation 112
3.7. System Test 113
3.8. Requirements Validation 118
3.9. Analysis of Emerging Challenges 118
3.9.1. Challenges of Design-Time Verification 119
3.9.2. Challenges of Testing 120
3.10. Scope of this Thesis 123
3.10.1. Introduction of Runtime Monitoring of Environment Assumptions 124
3.10.2. Research Questions of this Work 125
3.11. Summary 126

4. Solution Concept 129

4.1. Runtime Monitoring of Environment Assumptions 130
4.1.1. Integration in the System Development Process 131
4.1.2. Overview of Concept 132
4.1.3. Runtime Monitoring of Environment Assumptions by Example . . 136
4.2. Revisiting the Motivational Example 143
4.3. Requirements Elicitation and Analysis 149
4.3.1. Informal Specification of System Requirements 149
4.3.2. Formal Specification of System Requirements 154
4.4. Safety Analysis 156
4.4.1. HARA Analysis of the Revisited Motivational Example 156
4.4.2. “Safe Enough” for Autonomous Safety-Critical Systems 161
4.4.3. Extending Safety Requirements with Environment Assumptions . 163
4.4.4. Informal Specification of Extended Safety Requirements 165
4.4.5. Formal Specification of Extended Safety Requirements 167
4.5. System Design 174
4.5.1. Environment Model 174
4.5.2. Technical System Model 180
4.5.3. Design Time Verification 185

4.5.4. Analysis of the Environment Assumptions 186
4.5.5. Formal Definition of Environment Assumptions Monitors 188
4.6. System Implementation 191
4.6.1. Implementation of the System Model 191
4.6.2. Realization of the Environment Model 192
4.6.3. Realization of the Environment Assumptions Monitors 192
4.7. System Test 193
4.7.1. Testing the Implemented System 194
4.7.2. Testing Environment Assumptions Monitors 194
4.8. Requirements Validation 194
4.8.1. Environment Assumptions Validation via Runtime Monitoring . . 194
4.9. Summary 195

5. Case Studies 197
5.1. Case Study 1: Mobile Service Robot 198
5.1.1. Evaluation in the Operational Environment in the iserveU Project 198
5.2. Case Study 2: Automotive System Function for Accurate Vehicle Speed Estimation 203
5.2.1. Overview of Vehicle Speed Estimation Function 204
5.2.2. Requirements Elicitation and Analysis 205
5.2.3. Safety Analysis 206
5.2.4. System Design 213
5.2.5. System Implementation 225
5.2.6. System Test 225
5.2.7. Requirements Validation 229
5.3. Summary 231

6. Contributions with respect to Related Work 233
6.1. Obtaining Environment Assumptions 233
6.1.1. Manual Methods for Specification of Environment Assumptions . 234
6.1.2. Automated Methods for Obtaining Environment Assumptions . . 237
6.2. Assumptions in Verification and Validation Processes 241
6.2.1. Assumptions in Design-time Verification 241
6.2.2. Controller Synthesis and Environment Assumptions 243
6.2.3. Combining Design-time Verification with Runtime Validation . . . 245
6.3. Comparison of the Proposed Approach with Related Work 249
6.3.1. Comparison with the Manual Methods for Environment Assumptions Specification 249
6.3.2. Comparison with Automated Methods for Obtaining Environment Assumptions 251
6.3.3. Comparison with Methods that integrate Assumptions in Verification and Validation Processes 252
6.4. Summary 256

7. Summary and Conclusion **257**

7.1. Discussion of Results 257

7.1.1. Contributions 258

7.1.2. Limitations of this Thesis and Future Work 260

7.2. Summary 264

A. Addendum to Case Study 2: Vehicle Speed Estimation Function **267**

A.1. Design-time Verification of Extended Safety Requirements ESR2 - ESR4 267

A.2. Definition of Runtime Monitors for the Environment Assumptions of ESR2 - ESR4 269

A.3. System Test of the Environment Assumptions Monitors of ESR2 - ESR4 270

A.3.1. Scenario 1: Smooth Driving 270

A.3.2. Scenario 2: Dynamic Driving 272

Bibliography **275**

List of Figures

1.1. Overview of this Thesis Outline. 7

2.1. The System Development Process of ISO 26262 (cf. [Int11b]). 21
2.2. ISO 26262 - The Concept Phase in the System Development Process (cf. [Int11c]). 23
2.3. ISO 26262 - The Structure of the Safety Requirements through the System Development Process (cf. [Int11c]). 25
2.4. ISO 26262 - The Hardware Development Process (cf. [Int11f]). 26
2.5. ISO 26262 - The Software Development Process (cf. [Int11f]). 27
2.6. ISO/PAS 21448 - Visual Intuition of the Goal in the SOTIF System Development Process (cf. [Int19]). 31
2.7. ISO/PAS 21448 - Operative Flowchart of the SOTIF System Development Process (cf. [Int19]). 32
2.8. Visual Intuition of TCTL Formulae used in Uppaal (cf. [BDL04]). . . . 41
2.9. Visual Intuition of Testing. 58
2.10. Process of Model Checking (cf. [BK08]). 61
2.11. Visual Intuition of Model Checking. 62
2.12. Visual Intuition of Probabilistic Model Checking. 63

3.1. Mobile Service Robot: Physical Overview of the Motivational Example. . 77
3.2. Mobile Service Robot: Computation of the Collision Distance between the Robot and a Moving Obstacle. 79
3.3. Overall System Development Process. 81
3.4. Mobile Service Robot: A Visual Interpretation of the Safety Property. . . 101
3.5. Mobile Service Robot: Environment Model. 103
3.6. Mobile Service Robot: Technical System Model. 105
3.7. Mobile Service Robot: Declarations of Shared Variables in Uppaal. . . . 106
3.8. Mobile Service Robot: Coordinator Model in the Motivational Example. 107
3.9. Mobile Service Robot: Overall System Model and Communication between its Components in the Simulator Panel of Uppaal. 108
3.10. Mobile Service Robot: A Visual Representation of the Network Automaton's Initial State in Uppaal. 111
3.11. Mobile Service Robot: Spanning Tree Path Showing its Satisfied Safety Property. 112
3.12. Mobile Service Robot: Safety Property and Liveness Property Visualized in Uppaal. 113

3.13. Mobile Service Robot: A System Execution and the Corresponding Evaluation of the Safety Property Monitor M_ϕ during System Test. 116
3.14. Mobile Service Robot: State Space Explosion Problem Visualized in UPPAAL.120
3.15. Mobile Service Robot: A Visual Representation of the Initial System State during System Test. 121
3.16. Mobile Service Robot: Evaluation of the Safety Property Monitor M_ϕ showing the Safety Property ϕ fulfilled during System Test. 122
3.17. Mobile Service Robot: Evalution of the Safety Property Monitor M_ϕ showing the Safety Property ϕ disproved during System Test. 123

4.1. Overall System Development Process: the changes pertaining to this thesis' approach are highlighted in orange. 131
4.2. Illustration of Solution Concept With Example Values. 134
4.3. Mobile Service Robot: Application of RMEA to a System Trace with **valid Environment Assumption** ψ and **valid Safety Requirement** ϕ.140
4.4. Mobile Service Robot: Application of RMEA to a System Trace with **invalid Environment Assumption** ψ and **valid Safety Requirement** ϕ. 142
4.5. Mobile Service Robot: Application of RMEA to a System Trace with **invalid Environment Assumption** ψ and **invalid Safety Requirement** ϕ. 142
4.6. Physical Overview of the Revisited Motivational Example. 144
4.7. Overview of the Rules for Lane Changing in the Revisited Motivational Example. 145
4.8. Systematic Method for the Display and Comparison of the Reformulated System Requirements with the Original System Requirements. 149
4.9. Visual Intuition of the Movement Directions of Dynamic Obstacles with respect to the Mobile Service Robot. 158
4.10. Pattern for the Safety Requirements Specification extended with Environment Assumptions. 164
4.11. Representation of a Dynamic Obstacle Following the Robot in the Cartesian Coordinate System. 169
4.12. Construction Method for Runtime Monitors of Environment Assumptions. 190

5.1. Map of the Hospital Ward where the Mobile Service Robot has been evaluated in the iserveU Project (cf. [GRR$^+$16]). 199
5.2. Mobile Service Robot en route from Point (B) to Point (E) on the Hospital Ward (cf. [GRR$^+$16]). 201
5.3. Mobile Service Robot: Results of the Test Cases Evaluation on the Hospital Ward in the iserveU Project. 203
5.4. Vehicle Speed Estimation Function: High-level Overview (cf. [AVZR21]). 205
5.5. Vehicle Speed Estimation Function: Overview of Approach for Construction of the Functional Abstraction (cf. [AVZR21]). 213

5.6. Vehicle Speed Estimation Function: Visual Intuition of the Mechanism for Adequate GPS Data Selection (cf. [AZR20]). 220
5.7. Vehicle Speed Estimation Function: Environment Assumption Monitor for the GPS Data Error during the *Smooth Driving* Scenario. 227
5.8. Vehicle Speed Estimation Function: Test Data and Test Oracle of the Safety Requirement in the *Smooth Driving* Scenario (cf. [AZR20]). 228
5.9. Vehicle Speed Estimation Function: Environment Assumption Monitor for the GPS Data Error during the *Dynamic Driving* Scenario. 229
5.10. Vehicle Speed Estimation Function: Test Data and Test Oracle of the Safety Requirement in the *Dynamic Driving* Scenario (cf. [AZR20]). . . . 230

A.1. Vehicle Speed Estimation Function: Environment Assumption Monitor for the Road Slope during the *Smooth Driving* Scenario. 270
A.2. Vehicle Speed Estimation Function: Environment Assumptions Monitors for the Longitudinal Acceleration and the Lateral Acceleration during *Smooth Driving* Scenario. 271
A.3. Vehicle Speed Estimation Function: Environment Assumptions Monitor for the Road Slope during the *Dynamic Driving* Scenario. 272
A.4. Vehicle Speed Estimation Function: Environment Assumptions Monitors for the Longitudinal Acceleration and the Lateral Acceleration Environment during the *Dynamic Driving* Scenario. 273

List of Tables

2.1. SAE Automation Levels (cf. [SAE18]) vs. Autonomy Levels in Autonomous Systems (cf. [FMR+21]) 16
2.2. Overview of some Formalisms used for Modeling of Probabilistic Systems (cf. [Kat16]). 48

3.1. Mobile Service Robot: Requirements for Allowed and Forbidden System Actions in the System's Environment. 86
3.2. Mobile Service Robot: Requirements for the System's Normal Operation Mode. 87
3.3. Mobile Service Robot: Requirements for the System's Collision Avoidance Mode. 88
3.4. Mobile Service Robot: Hazard Analysis and Risk Assessment. 99
3.5. Mobile Service Robot: Detailed View in the Evaluation of the Safety Property Monitor M_ϕ with respect to the Visible Obstacles in the Robot's Environment. 117

4.1. Requirements for the allowed and forbidden system actions in the system environment: comparison between initial and revisited motivational example.151
4.2. Requirements for the normal operation mode of the robot: comparison between initial and revisited version the motivational example. 152
4.3. Requirements for the collision avoidance mode of the robot: a comparison between initial and revisited motivational example. 153
4.4. Mobile Service Robot - Hazard Analysis and Risk Assessment in the Revisited Motivational Example. 160
4.5. Mobile Service Robot - Safety Requirement extended with Environment Assumptions. 166
4.6. Mobile Service Robot: Informal and Formal Specification of the Safety Requirement extended with Environment Assumptions. 173
4.7. Mobile Service Robot: Verification of Extended Safety Requirement ESR2 expressed as Multi-objective Property. 186

5.1. Mobile Service Robot: Test Cases for the Evaluation on the Hospital Ward.202
5.2. Vehicle Speed Estimation Function: Hazard Analysis and Risk Assessment. 209
5.3. Vehicle Speed Estimation Function: Safety Requirement extended with Environment Assumptions. 211
5.4. Vehicle Speed Estimation Function: Informal and Formal Specification of the Safety Requirement extended with Environment Assumptions. 212

5.5. Vehicle Speed Estimation Function - Definition of Data Types for the Function Inputs and the Function Outputs in the Functional Abstraction. 214
5.6. Vehicle Speed Estimation Function - Definition of Data Types for the Intermediary Results in the Functional Abstraction. 215
5.7. Vehicle Speed Estimation Function - Definition of Application Parameters. 216
5.8. Vehicle Speed Estimation Function: Verification of Extended Safety Requirement ESR1 expressed as Multi-objective Property. 224

A.1. Vehicle Speed Estimation Function - Extended Safety Requirements ESR2 - ESR4 expressed as Multi-objective Properties. 268

style=list, title=Abbreviations

Chapter 1.

Introduction

1.1. Motivation 3
1.2. Goals and Contributions of this Thesis 5
1.3. Thesis Structure 6

1.1. Motivation

Automated safety-critical systems are deployed in highly safety- and mission-critical application domains, e.g., medical devices, aircraft flight control, vehicle control systems, or autonomous robots. Automated safety-critical systems are characterized by their continuous interaction with the physical world. The physical environment in which automated safety-critical systems are deployed is denoted henceforth as operational environment. Such systems are equipped with a set of sensors, which are used to perceive the operational environment of the system, as well as a set of actuators, with which the system may effect change on it. A failure in such systems may lead to mission failure with significant property damage or endanger human life. Due to their potential of causing harm, automated safety-critical systems benefit from a structured and rigorous development process, in which specific tasks are allocated for system verification and validation, in order to make that such systems are safe. Plan-driven development processes are considered suitable for the design and development of safety-critical systems (cf. [Som14a]). International standards focused on safety-critical and safety-related systems propose the V-model as a system development process for such systems (cf. [Int11b], [Int19]). For every phase in the development process, the system engineers define what artifacts are to be developed, the inputs needed from other development phases, the tools needed to develop the respective artifacts, as well as the guidelines and regulations to which these artifacts have to adhere. These guidelines and regulations may be internal to the organization or external, recommended by international standards or imposed by legislative bodies. Verification and validation are processes which accompany the actual development process of an automated safety-critical system, and can be applied in different phases of the system development and different abstraction levels. Depending on the abstraction level at which they are applied, the planning of the specific verification and validation tasks means deciding on the appropriate methods and tools with which the respective artifacts can be analyzed. Two well known categories of verification and validation methods are formal verification and testing.

Formal verification methods are used at design-time to obtain proof of the system's correctness with respect to a safety property specification. These methods rely on mathematical theories, e.g., logic or automata [Pel01], and allow the unambiguous description of the behavior of a system under analysis as well as the formal reasoning about this behavior with respect to a formal specification of the safety properties of interest (cf. [Luc21]). The system behavior is described by a formal model. This model is a mathematical abstraction of the system, that preserves only those system aspects which the system designers wish to analyze. Since automated safety-critical systems interact with the operational environment, design-time verification methods require also a model of the environment in order to carry out the verification process (cf. [FDA+18a]). During design of an automated safety-critical system, the system engineers define the operational design domain of the system under development. The operational design domain defines the general operational environment in which the system is intended to carry out its task as defined in the system requirements (cf. [KO19]). Some of the environmental characteristics are taken by the system engineers and included in the environment model. However, the environment model remains an abstraction of the actual system environment. In order to model the system reaction to the environmental characteristics specified in the environment model, the system engineers rely on a number of assumptions. These assumptions are used implicitly in the computations of the system, e.g., the assumption about the maximum obstacle velocity can be used to compute a collision avoidance strategy and subsequently plan a safe trajectory for an autonomous robot. These assumptions are rooted in the domain knowledge and years of experience which system modelers have of the system under development and its operational design domain, but are not defined explicitly. Other assumptions may not even be known to the system designers due to the system requirements and the definition of the operational design domain being incomplete.

Design-time verification is carried out by checking the system model together with the environment model against the formal specification of the system's safety requirements. Techniques used for verification at design-time are manual, e.g., deductive software verification, semi-automated, e.g., theorem proving, or fully automated, e.g., model checking. Since it is being applied on an abstract model of the system, the result of design-time verification is as good as the model of the system. Furthermore, design-time verification techniques, in particular model checking, suffer from scalability problems. In order to overcome its scalability issues, design-time verification is complemented by testing, which uses test oracles in the form of property monitors to check the actual system against the system requirements.

After passing all tests, the system is deployed in its operational environment. During the system operation, unpredicted events may occur and result in safety hazards. Given that the behavior of the system under analysis has been proven correct with respect to its safety requirements through design-time verification and that the actual system has passed all its tests, these hazards do not appear as consequence of malfunctioning system behavior. Instead, these hazards are caused by system's inability to deal with environment assumptions that have become invalid during system operation. This happens because there is no explicit definition of the respective environment assumptions that can be analyzed and considered in the design-time verification. Notice that the property monitors designed to check the system requirements are not suitable to verify for assumptions violation, since the assurance goal of a property monitor is to verify whether the system complies with the respective safety requirements. Given that there is no explicit definition of environment assumptions, the system designers cannot define and test specific monitors that can later be used to check for assumptions violations during system operation.

1.2. Goals and Contributions of this Thesis

In order to address these limitations, this thesis proposes an engineering approach which extends the quality assurance goals for automated safety-critical systems towards the verification and validation of environment assumptions. During system design, environment assumptions are explicitly defined and monitors are derived from the environment assumptions definition. During system testing, the quality assurance goals are to test the environment assumptions monitor as well as the monitor of the system's safety requirements. Requirements validation means both the validation of the system safety requirements as well as the validation of the environment assumptions.

The goal of this thesis is to define a structured engineering approach to ensure the safety of automated safety-critical systems in uncertain environments, by using environment assumptions defined explicitly at design-time to validate the design-time verification result during system operation. The approach is coupled with the system development process and uses the phases of the development process in order to produce the relevant artifacts for the definition of the environment assumptions. Thus, the contributions of this thesis are:

Method for Explicit and Formal Definition of Environment Assumptions. Based on a high-level description of the system under development, the functional system requirements are defined during the phase of requirements elicitation and analysis. The same high-level system description is used to carry out the safety analysis and derive safety hazards that could appear during system operation. Safety requirements and environment assumptions are defined that cover the respective safety hazards. The environment assumptions are used to extend the system safety requirements, which are then denoted as extended safety requirements. A requirements pattern is designed and applied in order to produce the informal specification of the extended safety requirements. Their formal specification is realized through TCTL for non-probabilistic systems and through a fragment of PCTL for probabilistic systems. The functional system requirements and the extended safety requirements serve as an input for the design of the technical system model and the environment model. Using the environment assumptions, these models are verified at design-time against the system safety requirements.

Method for Construction of Environment Assumptions Monitors. Although the environment assumptions have been explicitly and formally defined, for the construction of environment assumptions monitors to succeed, it is important to understand how their explicit definition of environment assumptions on the requirements level is mirrored at system design level. Environment assumptions build the interface between the technical system model and the environment model. An analysis of this interface is carried out and the environment assumptions are mapped on the technical system model and the environment model. The mapping of the environment assumptions on the technical system model is then projected on the implemented system. Further variable definitions are introduced to model the observations of the environment assumptions monitors in the system environment. The definition of the environment assumptions monitors combines variables of the implemented system and variables that model the monitor's observations in the system environment. On a concrete level, the monitors are defined as predicates in first-order logic.

Demonstration of the Applicability of the RMEA Concept on two Real-world Safety-critical Systems The applicability of the proposed safety engineering approach is demonstrated on two case studies: (i) a mobile service robot, commissioned to execute transportation tasks autonomously in an uncertain environment, and (ii) an automotive system function for estimation and display of a moving vehicle's speed on its instrument board.

1.3. Thesis Structure

Figure 1.1 illustrates the outline of this thesis and the relationships between the individual chapters.

Chapter 1: Introduction presents and motivates the problem of verification and validation of automated safety-critical systems in the absence of explicitly defined environment assumptions and introduces the contributions of this work.

Chapter 2: Fundamental Concepts and Approaches introduces concepts and approaches that are necessary to understand the following chapters of this thesis. It gives a detailed description of the development process for automated safety-critical systems. It then introduces several formalisms for modeling automated safety-critical systems and for formally specifying their safety properties, followed by a discussion of the fundamentals of verification and validation approaches used to guarantee the correctness of safety-critical systems with respect to a given property specification.

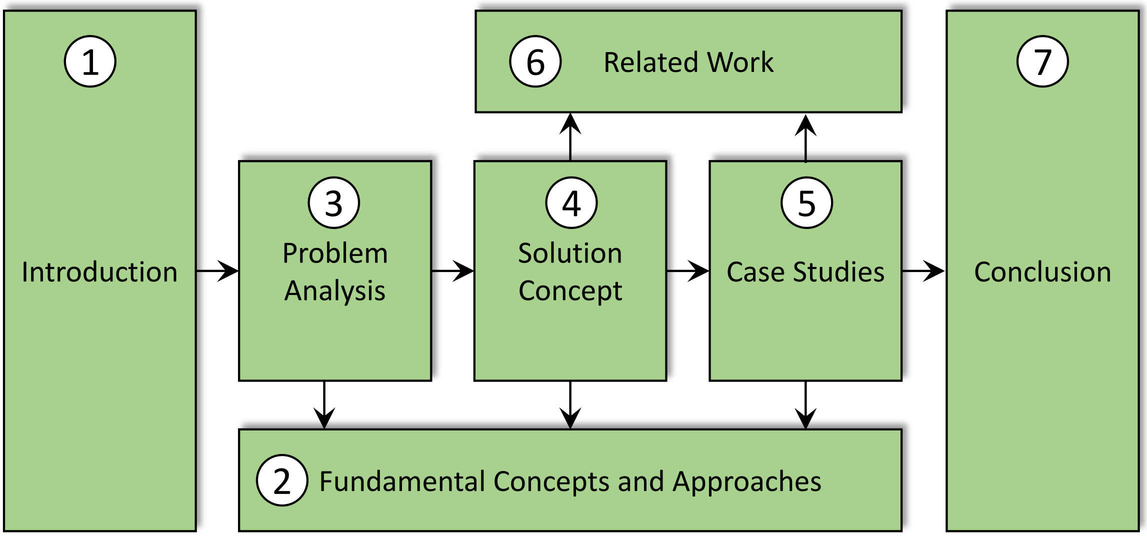


Figure 1.1.: Overview of this Thesis Outline.

Chapter 3: Problem Analysis introduces the motivational example - a mobile service robot commissioned to drive autonomously towards a given destination in an uncertain environment. Based on it, it carries out a problem analysis, which identifies the challenges of design-time verification and testing of automated safety-critical systems. These challenges lead to the definition of this thesis' research questions.

Chapter 4: Solution Concept presents an approach which addresses two of the research questions introduced through the problem analysis. The chapter unfolds what the application of this approach entails for every phase in the system development process based on the motivational example of the mobile service robot. This chapter provides a formal method for the definition of environment assumptions as well as method for the construction of environment assumptions monitors.

Chapter 5: Case Studies evaluates the approach presented in this thesis on two case studies: (1) the mobile service robot tasked to drive autonomously in an uncertain environment, and (2) an automotive system function for accurate speed estimation in driving vehicles. The evaluation is carried on based on simulations in the case of the mobile service robot and based on real-world data collected with a field testing platform of an automotive OEM partner.

Chapter 6: Related Work gives an overview of related work and compares it with the approach presented in this thesis.

Chapter 7: Conclusion summarizes this thesis and discusses further lines of research that can be derived from this work.

Note to the Readers of this Thesis

Interested readers of this thesis can read it the best way it suits them, in a linear manner or otherwise. However, the structure presented in Figure 1.1 is not random. Besides giving on overview of this thesis outline, Figure 1.1 can be interpreted also as a proposal on how to read this thesis. The backbone of this thesis is represented by Chapter 3, Chapter 4, and Chapter 5. These chapters have their theoretical foundations in Chapter 2, while Chapter 6 compares the concepts developed in Chapter 4 and Chapter 5 with other works. Chapter 1 and Chapter 7 represent, metaphorically speaking, the frame to the whole picture. It is worth considering reading the Chapters 3 to 5 and turning to Chapter 2 and respectively Chapter 6 only when more details on the underlying theoretical foundation of different concepts are needed or when comparison between this work and other research is looked for.

Chapter 2.

Fundamental Concepts and Approaches

2.1. Definitions of the Problem Domain 10
  2.1.1. Safety-critical Systems 10
  2.1.2. Functional Safety and Safety of the Intended Functionality 11
  2.1.3. Automated Systems versus Autonomous Systems 13
  2.1.4. Uncertain Environments 18
2.2. System Life Cycle and Development Process 19
  2.2.1. System Development Process according to ISO 26262 20
  2.2.2. System Development Process according to ISO/PAS 21448 29
2.3. Property Specification for Automated Safety-critical Systems 36
  2.3.1. Timed Computation Tree Logic 37
  2.3.2. Probabilistic Computational Tree Logic 40
2.4. System Modeling for Automated Safety-Critical Systems 44
  2.4.1. Timed Automata 45
  2.4.2. Short Overview of Modeling Formalisms for Probabilistic Systems 47
  2.4.3. Markov Decision Processes 49
2.5. Verification and Validation of System Properties in Automated Safety-critical Systems 53
  2.5.1. Testing 54
  2.5.2. Design-time Verification 58
  2.5.3. Runtime Verification 69
2.6. Summary 74

The goal of this chapter is to introduce several fundamental concepts and approaches on which this thesis builds upon. Section 2.1 starts this chapter by defining a few of the concepts that make up the research domain of this thesis. Section 2.2 gives an overview of the development process for automated safety-critical systems. Following this, in Section 2.4, several formalisms for modeling automated and autonomous safety-critical systems are presented in detail. In Section 2.3 the notion of safety and liveness property are introduced and several formalisms and languages designed to specify such properties are presented. Section 2.5 discusses the fundamentals of verification and validation approaches used to guarantee the correctness of safety-critical systems with respect to a given property specification. This chapter is concluded by a summary of the main concepts presented in Section 2.6.

2.1. Definitions of the Problem Domain

The problem domain of this thesis is sustained by several fundamental concepts. To begin with, Section 2.1.1 presents the concept of safety-critical system. In Section 2.1.2, the notions of functional safety and safety of the intended functionality are introduced. The concepts of automated and autonomous systems are discussed in Section 2.1.3, while Section 2.1.4 introduces the notion of uncertain environment.

2.1.1. Safety-critical Systems

Before introduction the concept of safety-critical system, the notions of system, subsystem and component must be defined. This is done in Definition 2.1.1, Definition 2.1.2, and respectively Definition 2.1.3.

Definition 2.1.1 - System
A *system* is considered to be a collection of elements which interact with each other following a set of specific rules in order to achieve a specific function or set of function within a specific environment. An element may include hardware equipment, a software program, or a human (cf. [ISO10]). ■

Definition 2.1.2 - Subsystem
A *subsystem* is a system in its own right, which is part of a larger system (cf. [ISO10]). In order to provide useful functionality it must be integrated with other subsystems in order to make a system. ■

Definition 2.1.3 - Component
A *component* is an entity with discrete structure, such as an assembly or software module, within a system considered at a particular level of analysis (cf. [ISO10]). A component is logically and technically separable from other parts in the system (cf. [Int11b]). A component may be either a hardware or a software component and can be divided in other components. Atomic level hardware components are denoted as hardware parts and atomic level software components are called also software units (cf. [Int11b]). ■

Notice that in literature, the notions of module, unit and component are used interchangeably and, depending on the context in which they are used, these notions may be defined as sub-elements of one another (cf. [ISO10]).

Definition 2.1.4 - Safety-critical System
A *safety-critical system* is a system "whose failure may endanger human life, lead to substantial economic loss or cause extensive damage to the environment" (cf. [Kni02]). In a safety-critical system, the safety of the system has a higher priority than other design objectives (cf. [Alu15]). ■

Safety-critical systems appear in a variety of application domains, such as commercial aircraft, medical care, nuclear power or weapons (cf. [Kni02]). Examples of safety-critical systems include pacemakers and insulin pumps in the medical care domain, autopilot systems in commercial aircraft and adaptive cruise control systems in the automotive industry. In parallel to the concept of safety-critical system, there is also the notion of safety-related system. The concept of safety-related system is introduced in IEC 61508, which is an international standard for the functional safety of electrical, electronic and programmable electronic (E/E/PE) equipment (cf. [Int97a]). A safety-related system is part of the system under analysis and implements the required safety functions needed to achieve or maintain a safe state of the system under analysis (cf. [Int97b]). At the same time, the safety-related system is designed to achieve, on its own or in combination with other E/E/PE safety-related systems or external risk mitigation measures, the safety integrity for the required safety function (cf. [Int97b]). Smith and Simpson [SS11] analyze the distinction between the concepts of safety-related system and safety-critical system and point out that these concepts are used in different situations. The concept of *safety-critical system* is used more often in situations in which a failure alone of the system in question leads to fatalities or increased exposure to risk for people. In contrast, the notion of *safety-related system* is used in a wider context, for systems in which a single failure is not necessarily critical, whereas the combination of coincident failures may lead to hazardous consequences (cf. [SS11]).

2.1.2. Functional Safety and Safety of the Intended Functionality

IEC 61508 works both as a stand-alone standard and as an umbrella standard that serves as a basis for the development of standards for specific sector industries and products (cf. [Int97a]). One industrial domain which developed its own standard using the foundations laid down by IEC 61508 is the automotive industry. ISO 26262 is the automotive standard, which is an adaptation of IEC 61508 in order to comply with the specific needs of the application domain of electrical and/or electronic (E/E) systems in road vehicles (cf. [Int11b]). This adaptation is reflected in all the phases of the system development process, which includes the design and development of the hardware and software system components, as well as their integration in the whole system and testing. Before defining the notion of functional safety, several terms related to the notion of safety must be defined. This is done in Definitions 2.1.6 to 2.1.11.

Definition 2.1.5 - Operational Situation
An *operational situation* is defined as a scenario that can occur during a system's life (cf. [Int11b]). ■

Definition 2.1.6 - Harm
Harm is defined as physical injury or damage to the health of persons (cf. [Int11b]). ■

Definition 2.1.7 - Severity
Severity is defined an estimate of the extent of harm to one or more persons which can occur in a potentially hazardous situation (cf. [Int11b]). ■

Definition 2.1.8 - Risk and Unreasonable Risk
Risk is defined as the combination of the probability of occurrence of harm and the severity of that harm. *Unreasonable risk* is defined to be a risk that is considered to be unacceptable in a certain context and according to valid societal moral concepts (cf. [Int11b]). ■

Definition 2.1.9 - Residual Risk
Residual risk is the risk which remains after the safety measures have been deployed (cf. [Int11b]). ■

Definition 2.1.10 - Malfunctioning Behavior
A *malfunctioning behavior* of a system under analysis is the failure or the unintended behavior of the system with respect to its design intent expressed in the system requirements (cf. [Int11b]). ■

Definition 2.1.11 - Hazard
A *hazard* is a potential source of harm caused by malfunctioning behavior of the system under analysis (cf. [Int11b]). ■

Definition 2.1.12 - Hazardous Event
A *hazardous event* is defined as a combination of a hazard and an operational situation (cf. [Int11b]). ■

Once the relevant notions related to safety have been introduced, the concept of functional safety can be defined, which is done in Definition 2.1.13.

Definition 2.1.13 - Functional Safety
Functional safety is defined as the absence of unreasonable risk which appears due to hazards caused by the malfunctioning behavior of E/E systems as well as the software installed on such systems (cf. [Int11b]). ■

According to Definition 2.1.13, the malfunctioning behavior of both hardware components and software components can lead to a loss of the system's functional safety. Notice that functional safety refers only to the cases when system failures occur. A failure of the system under analysis is equivalent to the inability of the system to work as required. Functional safety is therefore also concerned with methods and measures for mitigating the risks which occur due to system failure.

However, even in the absence of system failure, the system under analysis may be unable to work as specified in the system requirements. This can happen due to the appearance of unforeseen events in the environment in which the system under analysis operates. Such events expose functional insufficiencies in the system under analysis, which are performance limitations in the implementation of the specified system behavior. To give some examples of performance limitations, consider the example of an adaptive cruise control (ACC). Following limitations in the implementation of the system under analysis can be exposed by events which occur in the environment: (1) incomplete perception of the scene in which the vehicle is situated due to heavy rain, (2) limitation of the ACC decision algorithm in the ego vehicle due to abrupt braking of the leading vehicle, or (3) insufficient performance of the actuators in the ego vehicle due to skidding on a wet road surface. In order to address this type of limitations, the system engineers have to ensure not only the functional safety of the system under analysis but also the safety of its intended functionality. Definition 2.1.14 describes the notion of safety of the intended functionality.

Definition 2.1.14 - Safety of the Intended Functionality
The *safety of the intended functionality (SOTIF)* is defined as the absence of unreasonable risk which appears due to hazards resulting from functional insufficiencies or through reasonably foreseeable misuse of the system under analysis by the system users (cf. [Int19]). ■

2.1.3. Automated Systems versus Autonomous Systems

Autonomous systems have permeated various application domains in the last decades, ranging from health-monitoring devices to autonomous robots and vehicles. From an etymological point of view[1], the adjective *autonomous* originates in the Greek language and is the composition of two particles: *autos* which means “self” and *nomos* which means “law”. Thus, any entity described as being autonomous is considered to be subject to its own laws or rules and to be able to self-govern itself according to those laws. Various researchers give their own definition of an autonomous system. Autonomous systems are considered to be systems that are able to observe and orient themselves in their environment as well as make and execute decisions in order to achieve their goal without any outside assistance from humans or from any other systems (cf. [HMA03], [CHMS06]). Having control over their own actions and over their own state, autonomous systems are able to decide, based on their own state and on their sensing of their environment, what tasks, at which point in time and in which order need to be carried out (cf. [FDW13]). Furthermore, autonomous systems are systems which are able to change their behavior in response to unforeseen events in their environment (cf. [WS05]). Various technologies which help improve the self-awareness of autonomous systems along with the expansion of artificial intelligence (AI) theory bring further dimensions to the definition of autonomous systems. Besides being able to perform tasks independently of the supervision of a human operator, AI-based autonomous systems can exhibit behaviors that allow them to evolve and gain certain levels of self-X features, e.g., self-adaptation or self-healing (cf. [Xu21]). Furthermore, when reacting to unanticipated events in their environment, the results may not be deterministic (cf. [Xu21]). Definition 2.1.15 describes autonomous systems as they are seen and used throughout this thesis.

Definition 2.1.15 - Autonomous System
An *autonomous system* is a system which is able to observe its environment, find its orientation in it, make and execute decisions without any external (human) supervision (cf. [HMA03], [FDW13]) as well as change its behavior in reaction to any unanticipated events in the environment (cf. [WS05]), possibly with nondeterministic results (cf. [Xu21]). ■

[1] https://www.etymonline.com/word/autonomous

There are various situations in which autonomy is worth considering: (1) physically challenging tasks for humans due to its extensive duration or due to a large number of repetitions, (2) necessity for a faster reaction time than that of a human, and (3) hostile or remote environments inaccessible tu humans in which the tasks are to be performed (cf. [FDW13]). In relation to the notion of autonomous systems there is also the concept of automated system. Yet, the two notions are very different from each other. Definition 2.1.16 describes automated system as they are seen and used henceforth in this thesis.

Definition 2.1.16 - Automated System
An *automated system* is a system which is able to perform well-defined tasks and to produce deterministic results, while relying on a fixed set of rules and algorithms (cf. [Xu21]). ■

According to the Definitions 2.1.15 and 2.1.16, every autonomous system is an automated system but not every automated system is an autonomous system. Thus, the set of autonomous system is a subset of the set of automated systems. Throughout this thesis, the systems taken under analysis are automated safety-critical systems, which interact continuously with the operational environment in which they are deployed.

Notice that Definition 2.1.15 describes the maximum level of autonomy. There are, however, several degrees of autonomy (cf. [DF20], [FMR$^+$21]). The taxonomy introduced in [DF20] follows closely the PACT system for pilot authorization for control of tasks developed for aerospace scenarios (cf. [BTFM08]), while the classification in [FMR$^+$21] is closer to some extent to the SAE automation levels defined in the SAE J3016 standard (cf. [SAE18]). Each degree of autonomy defined in [FMR$^+$21] is a combination between the responsibilities by autonomous system and the responsibilities of a human operator. A comparison of the two taxonomies introduced in [FMR$^+$21] and [SAE18] is given in Table 2.1. In comparison to the six SAE automation levels, Fisher et al. introduce an additional level for low autonomy which is situated between SAE level 0 (no automation) and SAE level 1 (driver assistance) (cf. [FMR$^+$21]).

SAE level 0 corresponds to autonomy level 0 in Fisher's taxonomy (cf. [FMR$^+$21]). At SAE level 0, the driver is required to perform all aspects of the driving task. There may be safety mechanisms installed in the vehicle to provide warnings or support, e.g., forward collision warning system (cf. [SAE18]).

SAE level 1 corresponds to autonomy level 2 and comprises driving assistance systems which support the driver in controlling the vehicle. The driving assistance system executes within certain limits either the lateral motion control or the longitudinal motion control, while the human driver is in charge of permanently performing the remainder of the driving tasks, that are not carried out by the driving assistance system. As an example, if the driving assistance system is in charge of longitudinal motion control, then the driver is responsible for executing lateral motion control (cf. [SAE18]). The driver is also responsible for supervising the operation of the system and monitoring its environment. Furthermore, the driver is also required to be able to determine when the engagement or disengagement of the automated driving system is necessary (cf. [SAE18]). As a consequence, the driver represents the only fallback for the driving assistance system, and is required to be able to take over immediately complete control of the vehicle at any time, should a hazardous event occur (cf. [SAE18]). Examples of automated systems at SAE level 1 are ACC and lane keep assist (LKAS) systems.

SAE level 2 refers to partial driving automation and corresponds to Fisher's autonomy level 3, which represents systems with partial autonomy (cf. [SAE18], [FMR$^+$21]). Partially automated driving systems perform more tasks on their own, e.g., executing both longitudinal and lateral motion control of the vehicle for a certain period of time or in specific situations described in the ODD (cf. [SAE18]). The human driver can take the hands off the steering wheel for a short time while the automated driving system is activated. However, the driver is required to constantly to supervise the automated system and monitor its environment, and be prepared to intervene immediately in case of a hazardous event (cf. [SAE18]). It is worth noticing that in the autonomy levels 2 and 3, which correspond to the SAE levels 1 and 2, the system permits activation by a human operator, irrespective of whether the current environment conditions are part of its ODD or not (cf. Definition 2.2.7). This is because the automated driving system is not required to monitor its own environment. Instead, it is the responsibility of the human driver to monitor the system environment and determine whether the automated driving system can be activated or not. An example of a SAE level 2 automated driving system is the Tesla Autopilot system mounted in Tesla vehicles (cf. [Mor21]).

SAE level 3 describes conditional driving automation and is considered to be equivalent to the Fisher's autonomy level 4, which is also denoted conditional autonomy (cf. [FMR$^+$21]). An automated driving system with SAE level 3 is required to be able to execute the entire driving task independently and without human intervention for a limited period of time and only in certain situations described in the system's ODD (cf. [SAE18]). The vehicle driver is allowed to turn away temporarily his attention from the driving task and from the surrounding system environment. Even though the driver can turn away his attention from the driving task for a short period of time, he is still required to be receptive to the requests to intervene and act as the fallback mechanism for the automated driving system (cf. [SAE18]). Automated driving systems of SAE level 3 and higher are required to monitor their own environment and determine if the environment conditions are inside their respective ODD. If it determines that its ODD limits are about to be exceeded, then the automated driving system is required to issue a timely request for intervention to the vehicle driver (cf. [SAE18]). The automated driving system disengages immediately upon the driver's request or after a certain amount of time if the driver does not respond to the request to intervene (cf. [SAE18]). An example of an SAE level 3 automated driving system is the traffic jam assistant manufactured by Mercedes-Benz (cf. [Pet20], [dpa22]).

SAE level 4 refers to high driving automation and is regarded as equivalent to the autonomy level 5 in Fisher's taxonomy, which is called high autonomy (cf. [FMR$^+$21]). Since it is required to monitor its own environment, an automated driving system of SAE level 4 allows engagement only when the environment conditions satisfy the constraints imposed by the ODD definition (cf. [SAE18]). Similar to SAE level 3, as long as the environment conditions satisfy the ODD definition, the automated driving system is required to execute the entire driving task independent of any human intervention.

Table 2.1.: SAE Automation Levels (cf. [SAE18]) vs. Autonomy Levels in Autonomous Systems (cf. [FMR+21])

SAE Automation Level	**Narrative Description**	**Autonomy Level**	**Narrative Description**
No automation (SAE Level 0)	The driver is in charge of all aspects of the driving task, even though active safety mechanisms installed in the vehicle may provide support.	No autonomy (Level 0)	The operator is in charge of all tasks.
-	-	Low autonomy (Level 1)	Straightforward non-trivial tasks are performed by the system. The operator is not required to take over operation
Driver assistance (SAE Level 1)	The system executes either the longitudinal or the lateral motion control task within ODD limits, but not both at the same time. The driver performs the remainder of the driving task, monitors the system and the driving environment, is prepared to intervene immediately and execute the fallback if needed/desired. The system disengages immediately on driver's request.	Assistance systems (Level 2)	The operator is assisted by automated systems. The operator remains in control or is ready to take back control of the entire driving task at any time.
Partial automation (SAE Level 2)	The system executes both the longitudinal and the lateral motion control tasks within ODD limits. The driver performs the remainder of the driving task, monitors the system and the driving environment, is prepared to intervene immediately and execute the fallback if needed/desired. The system disengages immediately on driver's request.	Partial autonomy (Level 3)	The automated control system takes full control of the system. The operator remains engaged, monitors the operation and is prepared to intervene immediately.

SAE Automation Level	**Narrative Description**	**Autonomy Level**	**Narrative Description**
Conditional automation (SAE Level 3)	When within ODD limits, the system executes the entire driving task without human intervention. The system monitors its environment and determines if the ODD constraints are satisfied. The driver is expected to respond appropriately to a timely request to intervene from the system. The system disengages immediately upon driver's request or a certain amount of time after a request to the driver to intervene.	Conditional autonomy (Level 4)	The automated control system has full control of the operation during specified tasks. The operator can turn safely his attention away, but must be ready to take over control if needed.
High automation (SAE Level 4)	Within ODD limits, the system executes the entire driving task without human intervention. The system monitors its environment and determines if the ODD constraints are satisfied. The driver is not expected to respond to a request to intervene from the system. If the driver does not respond, then the system executes the fallback and brings the vehicle in a safe state. The system disengages after it has reached a safe vehicle state or if the driver executes the driving task.	High autonomy (Level 5)	The automated control system is capable of performing all planned functions under given conditions. The operator can leave the system safely to work by itself.
Full automation (SAE Level 5)	The system performs all aspects of the driving task under all aspects of the driving task, under all roadway and environmental conditions that can be managed by a human driver.	Full autonomy (Level 6)	The system can perform all its intended tasks on its own. The operator is not required to intervene at any time.

In case a hazardous event occurs or the ODD limits are exceeded, the system requests the intervention of the vehicle driver (cf. [SAE18]). In contrast to SAE level 3, there is no expectation for SAE level 4 systems that the driver responds to a request to intervene. In case the driver does not respond to the system's request to intervene, then the automated driving system is required to execute the fallback and achieve a safe state for the vehicle.

In case he chooses to intervene, the driver requests the automated driving system to disengage and becomes its fallback mechanism in order to bring the vehicle in a safe state (cf. [SAE18]). In case the driver does not intervene, the automated driving system is required to be able to execute the fallback in a timely manner and, automatically and on its own, bring the vehicle in a safe state (cf. [SAE18]). At the moment, there exists no market-ready automated driving system that complies with SAE level 4 (cf. [Sch21]).

SAE level 5 represents the highest level of automation, denoted as full driving automation, and corresponds to autonomy level 6 in Fisher's taxonomy (cf. [FMR$^+$21]). In comparison to SAE level 4, the performance and execution of the driving task by an automated driving system at SAE level 5 is not ODD-specific anymore (cf. [SAE18]). This means that the automated driving system permits engagement under all on-road conditions that are manageable by a human driver (cf. [SAE18]). As with SAE level 4, no automated driving system exists on the market which complies with the requirements for SAE level 5.

2.1.4. Uncertain Environments

Automated systems have usually been deployed in known, restricted environments. The main characteristic of such environments is that their structure, configuration as well as their dynamics are defined a priori during system design. An automated system working under the premise that its operation environment is completely defined at design-time is said to be working under the *closed-world assumption* (cf. [Mau19]). An example of an automated system deployed in a closed environment is an industrial robot deployed in a production line of a car manufacturer. These robots are usually restricted in an area with predefined dimensions, separated by walls from the rest of the factory floor. Furthermore, there may be clear rules of interaction between the human workers and the industrial robots, e.g., if the industrial robot is powered on, then no human workers are allowed to enter the restricted area.

However, automated systems are used also in less structured environments, such as roadways, hospitals or households. The main characteristic of such environments is that they are open and dynamic, which leads to some features of the environment being known only during the system operation. An automated system working under the premise that its operational environment cannot be fully specified at design-time is said to be working under the *open-world assumption* (cf. [Mau19]). Such an automated system is sometimes also referred to as an open context system, as the unpredictability and inherent complexity of the operational environment make it impossible for the system designers to give a complete specification of it at design-time (cf. [BH20]). When working under the open-world assumption, the interaction between an automated system with its operational environment is an important source of uncertainty. Therefore it becomes important to pinpoint the key features which trigger this uncertainty and define the notion of uncertain environment for automated systems. Definition 2.1.17 gives a description of how the concept of uncertain environment is understood throughout this thesis.

Definition 2.1.17 - Uncertain Environment of Automated Systems
For an automated system, an *uncertain environment* is an unstructured and heterogeneous part of the physical world, with complex and unpredictable dynamics, e.g., moving objects or other entities which exert changes on the physical world. Any a priori knowledge retained about such an environment is necessarily:

- *incomplete*, e.g., temporary features of the environment are omitted,
- *inaccurate*, e.g., spatial relations have changed in the environment since the a priori knowledge was gathered, and
- *approximate*, e.g., metric information in the environment may be imprecise.

■

To give an example of an uncertain environment and the respective a priori knowledge of this environment, consider an autonomous robot which is commissioned to drive to a given destination. Prior to the start of its mission, the robot has stored a map of the operation environment in which it is supposed to drive. This map represents the a priori knowledge retained by the robot about its environment. However, details and temporary features may be omitted from the map, so the map is incomplete. Furthermore, spatial relations between entities may have changed in the operation environment since the map was built, which renders the map inaccurate. Moreover, the metric information of the map may be imprecise, which makes the map an approximate one.

2.2. System Life Cycle and Development Process

Automated systems are often used in highly safety- and mission-critical applications. Failure of the software responsible for the control of automated systems could lead to mission failure and endanger human life. Due to their complexity and their critical nature, a structured and rigorous system development process with clearly specified tasks for verification and validation becomes paramount for the development of automated systems. System development models and development processes are used to achieve a structured and controllable effort during development of systems (cf. [SLS14a]). For automated safety-critical systems, e.g., speedometers in vehicles, and autonomous systems, e.g. mobile service robots, a plan-driven development process is appropriate (cf. [Som14a]). International standards concerned with functional safety of safety-related and safety-critical systems recommend the use of the V-model for the development of such systems in their respective application domains.

This section gives an overview of the system life cycle and the development process for automated and autonomous safety-critical systems, as presented in two international industry standards, ISO 26262 and ISO/PAS 21448. ISO 26262, first published in 2011 and revised in 2018, is the adaptation of IEC 61508 for the automotive industry, in order to comply with the specific requirements of E/E systems in road vehicles (cf. [Int11b]). The ISO 26262 standard provides a full automotive safety life cycle and describes the automotive development process in detail, which is briefly presented in Section 2.2.1. The ISO/PAS 21448 standard extends the development process of ISO 26262 and provides mechanisms to guarantee the safety of the intended functionality in the absence of system faults (cf. [Int19]). These extensions are discussed in Section 2.2.2.

2.2.1. System Development Process according to ISO 26262

The development process in ISO 26262 is depicted in Figure 2.1. Under ISO 26262, the system under development is considered to be a series production road vehicle (cf. [Int11b]). ISO 26262 differentiates between development at system level, hardware level, and software level and defines specific development processes for each of them. Although the three development processes are different from each other, the development activities at software level are coordinated with the development activities at hardware level and with the activities carried out at system level. The development process proposed in ISO 26262 begins at system level with the concept phase which lays down the functional and non-functional requirements of the system as well as the system boundaries and the dependencies of the system with its environment. During the concept phase, a hazard analysis and risk assessment is carried out in order to identify and categorize the hazards which could occur due to malfunctioning behavior of the system. The results of the concept phase are a functional concept and a functional safety concept. The functional concept describes the system functional and non-functional requirements, while the functional safety concept contains the functional safety requirements of the system under development.

Having defined the functional concept and the functional safety concept for the system under development, the planning of the development at system level is initiated. The purpose during initiation of system development is to define a development plan and a functional safety plan using the functional concept and respectively the functional safety concept as input. These plans determine the development activities at system level and plan the respective functional safety activities during the phases of the system development.

During development at system level, technical safety requirements are derived from the functional safety requirements specified during concept phase. The system design begins once the development plan is defined and the technical safety requirements are specified. During system design, the logical system architecture is defined based on the functional requirements defined for the system under development. The logical system architecture described the network of the system functions, the interfaces of the individual functions and the communication between the functions for the entire vehicle or for a subsystem of the vehicle (cf. [SZ16]).

The specification of the concrete technical system architecture is derived on the basis of the logical system architecture. The technical safety requirements are allocated to hardware and software (cf. [Int11d]), and the technical system architecture specifies which of the system functionality is to be implemented by software and which functionality is

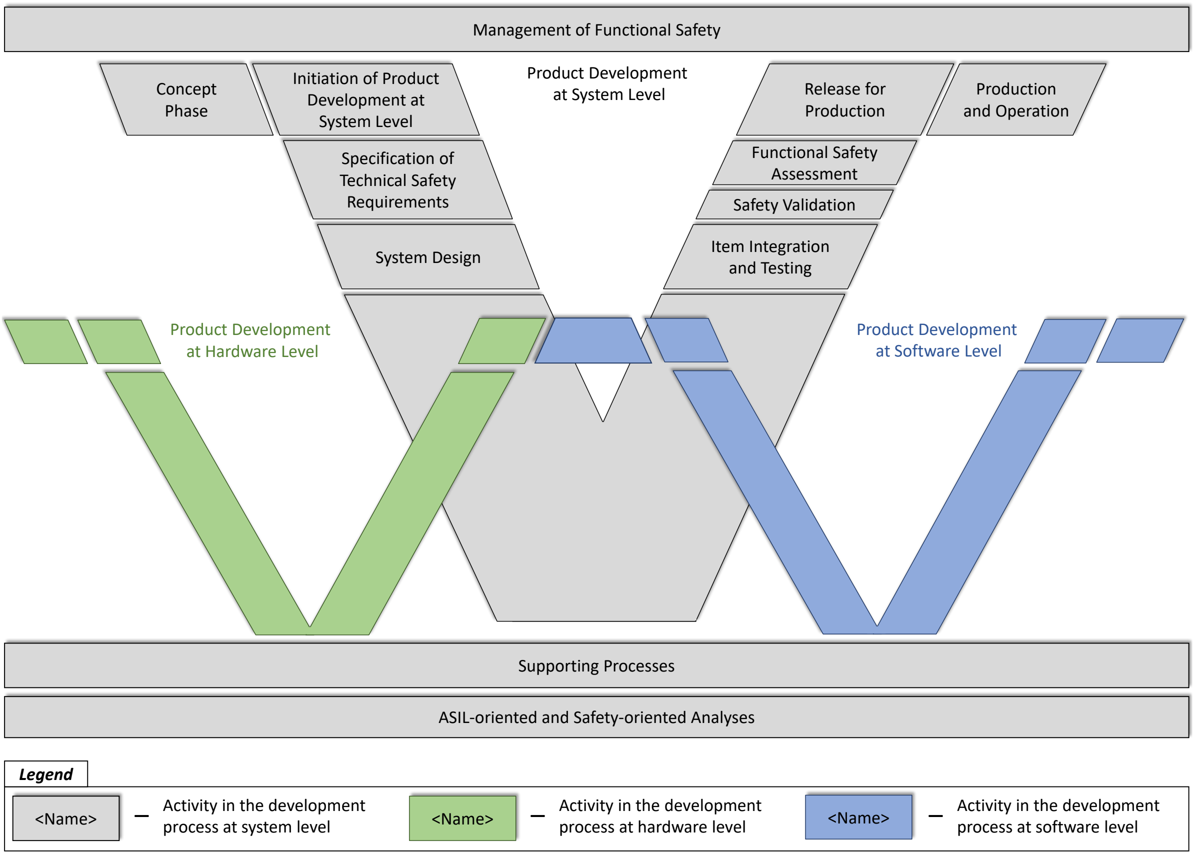

Management of Functional Safety
Concept Phase
Initiation of Product Development at System Level
Product Development at System Level
Release for Production
Production and Operation
Specification of Technical Safety Requirements
Functional Safety Assessment
Safety Validation
System Design
Item Integration and Testing
Product Development at Hardware Level
Product Development at Software Level
Supporting Processes
ASIL-oriented and Safety-oriented Analyses
Legend
<Name> – Activity in the development process at system level
<Name> – Activity in the development process at hardware level
<Name> – Activity in the development process at software level

Figure 2.1.: The System Development Process of ISO 26262 (cf. [Int11b]).

to be realized in hardware (cf. [SZ16]). The technical safety requirements are further refined and additional requirements with respect to the hardware-software interface are added to the requirements catalog. The phase of system design is applied in an iterative manner to every subsystem in the system under development. The phase of system design is followed by the development at hardware level and respectively software level.

The implemented software components are checked for correctness by component or module tests. The tested software components are integrated to subsystems of the software system and then the entire software system, which is subsequently subject to integration tests. Three major integration phases follow after the software integration test.

The first phase is the hardware-software integration, which takes place after the software integration test is completed. In this phase, the software system is installed on the corresponding electronic control units (ECUs), and the ECUs are calibrated so that they are functional. Tests are carried out to verify the hardware-software interface, while diagnostic coverage is used to measure the effectiveness of the implemented safety mechanisms with respect to specific predefined hardware metrics.

The second major integration phase is the system integration. The ECUs are integrated with other E/E components, e.g. sensors and actuators, and with elements of other technologies, e.g., mechanical components (cf. [Int11d]). The integrated system represents a subsystem of the vehicle and is checked for correct behavior in the integration test of the system (cf. [SZ16]).

The last phase of integration is vehicle integration, in which the different subsystems are assembled in the vehicle. After the integration at the vehicle level is finished, safety validation is carried out to provide evidence of functional safety with respect to the defined safety goals. The tests carried out during safety validation take into consideration the safety goals, the functional safety requirements and the intended use cases. The phase of safety validation is followed by an assessment of the vehicle's functional safety, which is carried out by an independent third party, e.g., a certification body. Once the functional safety assessment is successful, the system is released for production (cf. [Int11d]).

Engineering Functional Safety in ISO 26262

The end objective of a system development process carried according to ISO 26262 is to demonstrate a certain level of safety for the system under development. In order to achieve this objective, the process proposed in ISO 26262 aims to reduce the unreasonable risks to which the system under development may be subject later during its operation and build in mechanisms which help manage these risks, either by avoiding them or by mitigating them. The process begins with the concept phase, which produces as artifacts a functional concept as well as functional safety concept of the system under development. An overview of the concept phase is shown in Figure 2.2.

The concept phase begins with the item definition, in which the system functional and non-functional requirements as well as the system boundaries, the system interfaces and the assumptions regarding the interactions with other systems are defined. The system requirements describe the purpose and functionality of the system, its operating modes and states, the operational and environmental constraints, and include legal requirements extracted from regulations and standards (cf. [Int11c]). The safety life cycle is then initiated with the goal to determine whether the system under development is a new system to be developed or is an existent system, that undergoes modifications. If the system under development is newly to be developed then the next step is hazard and risk assessment. However, if the system under development is an existent system that is subject to modifications due to changes in the system requirements, then an impact analysis is performed in order to find out how the required modifications reflect on the relevant phases of the system development and on the safety plan of the system (cf. [Int11c]).

The hazard analysis and risk assessment (HARA) is performed in order to identify and categorize the hazards which could occur due to system malfunctions. For the hazard identification, a situation analysis is performed which takes into account the operating modes and operational situations in which a system failure will lead to a hazardous event, for the cases when the system is correctly used as well as when it is incorrectly used in a foreseeable manner (cf. [Int11c]). Hazard identification can be carried out systematically using techniques such as fault tree analysis (FTA) [BRB13], failure mode and effects analysis (FMEA) [DAR+15], or hazard and operability study

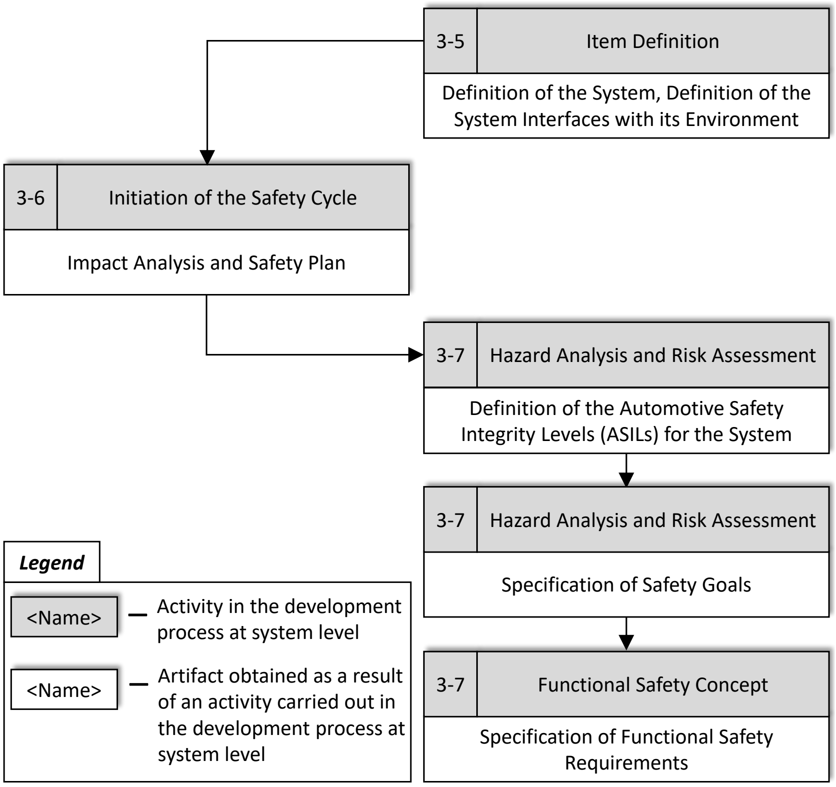

3-5
Item Definition
Definition of the System, Definition of the System Interfaces with its Environment
3-6
Initiation of the Safety Cycle
Impact Analysis and Safety Plan
3-7
Hazard Analysis and Risk Assessment
Definition of the Automotive Safety Integrity Levels (ASILs) for the System
3-7
Hazard Analysis and Risk Assessment
Specification of Safety Goals
3-7
Functional Safety Concept
Specification of Functional Safety Requirements
Legend
<Name> – Activity in the development process at system level
<Name> – Artifact obtained as a result of an activity carried out in the development process at system level

Figure 2.2.: ISO 26262 - The Concept Phase in the System Development Process (cf. [Int11c]).

(HAZOP) [DFVA09]. Hazard identification is followed by a risk assessment, which does a classification of the hazardous events according to which are the most serious and the most likely to occur. This is formulated in terms of risk, which is the combination of the probability of occurrence of harm and the severity of that harm (cf. Definition 2.1.8). In turn, the probability of occurrence of harm is determined as a combination between the probability of exposure to a hazardous event and the probability that the driver gains sufficient control over the hazardous event so as to avoid the specific harm. Using three parameters - severity of harm (S), exposure (E), and controllability (C) - a risk matrix is built which is used to establish the automotive safety integrity level for each hazardous event identified for the system under development.

Definition 2.2.1 - Automotive Safety Integrity Level (ASIL)
An *automotive safety integrity level (ASIL)* represents one in four levels associated with the safety requirements for a system under development and the safety measures needed to be applied in order to avoid unreasonable residual risk. There are four levels, with D being the highest and A being the lowest safety integrity level (cf. [Int11b]). ■

Having determined the ASIL of a system under development, the safety goals associated with it are specified. A safety goal is a top level requirement, which is specified for each hazardous event identified in the hazard analysis, that has an ASIL allocated to it. Notice that there is a *n-to-n* relationship between safety goals and hazards, i.e. one safety goal can be related to several hazards and, at the same time, one hazard can be covered by several safety goals (cf. [Int11b]).

The functional safety concept is defined based on the system safety goals and consists of a catalog of functional safety requirements. These requirements specify various safety measures for fault detection, fault tolerance, failure mitigation, which can be applied or implemented in the system in order for the system to comply with its safety goals (cf. [Int11c]). An example of a safety measure is FTA, which can be used for fault detection.

Definition 2.2.2 - Safety Measure
A *safety measure* represents an activity or a technical solution that can be realized in order to avoid or control systematic failures and to detect or control random hardware failures, or mitigate their potentially harmful effect (cf. [Int11b]). Safety measures can include safety mechanisms. ■

Definition 2.2.3 - Safety Mechanism
A *safety mechanism* is a technical solution whose purpose is to detect faults or control failures in order to achieve or maintain a safe state of the system under development (cf. [Int11b]). ■

Along with the functional safety concept, the concept phase creates also a preliminary architectural design of the system under development. Based on this preliminary design, the ASIL of the entire system under development can be decomposed following its hierarchical structure. Thus, the safety-related subsystems and components in the system under development have an ASIL associated to it.

During the development at system level, the functional safety requirements are refined into technical safety requirements, which specify how the functional safety concept is to be implemented in the system under development. This includes defining the response of the system to stimuli which affect its safety goals, i.e. for each system operating mode and system state, the relevant combination of stimuli and the possible system failure are defined (cf. [Int11d]). During development at hardware and software level, the hardware and software safety requirements are derived from the technical safety requirements. The structure of the safety requirements throughout the system development process is shown in Figure 2.3.

Hardware Development Process in ISO 26262

The hardware development process according to ISO 26262 is depicted in Figure 2.4. The process begins with planing the activities for the development at hardware level. These activities consist of the hardware implementation of the technical safety requirements, the analysis of the hardware faults and their effects, as well as the coordination of the hardware development with the software development (cf. [Int11e]).

Before the hardware design can begin, the hardware safety requirements are defined based on the technical safety requirements and the technical system architecture defined

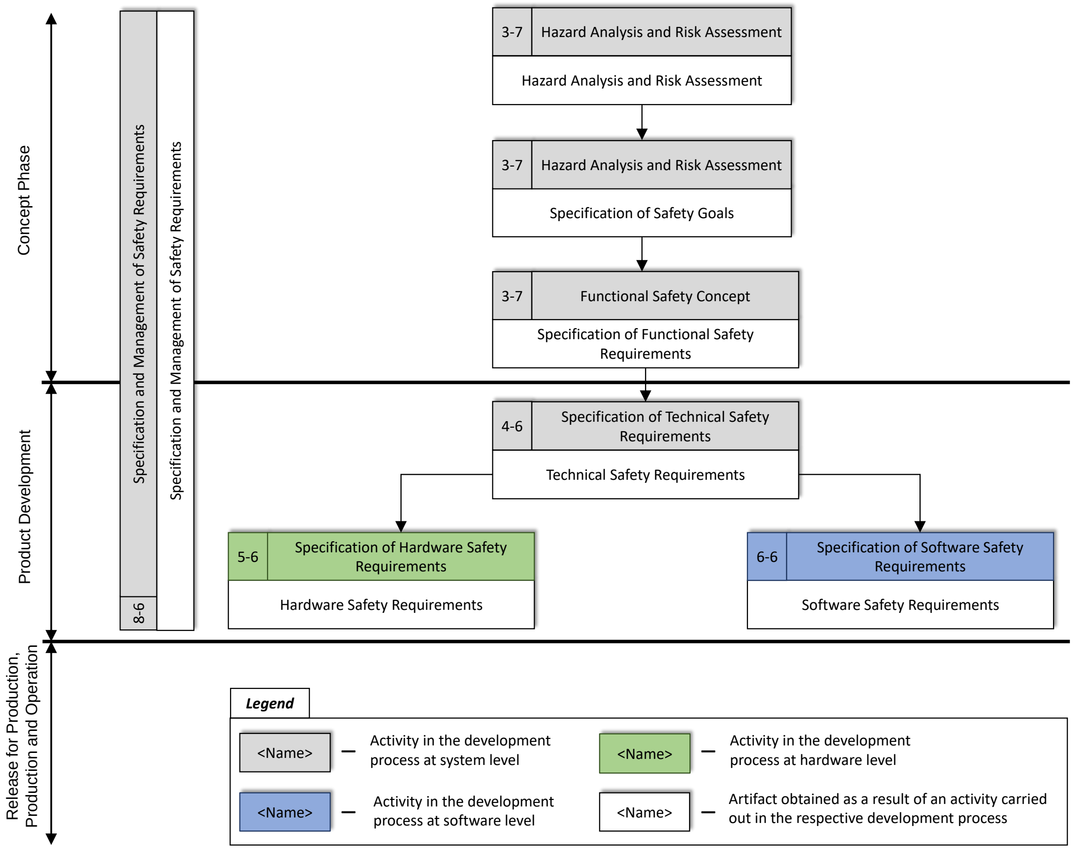

Concept Phase
Product Development
Release for Production, Production and Operation
Specification and Management of Safety Requirements
Specification and Management of Safety Requirements
8-6
3-7
Hazard Analysis and Risk Assessment
Hazard Analysis and Risk Assessment
3-7
Hazard Analysis and Risk Assessment
Specification of Safety Goals
3-7
Functional Safety Concept
Specification of Functional Safety Requirements
4-6
Specification of Technical Safety Requirements
Technical Safety Requirements
5-6
Specification of Hardware Safety Requirements
Hardware Safety Requirements
6-6
Specification of Software Safety Requirements
Software Safety Requirements
Legend
<Name> – Activity in the development process at system level
<Name> – Activity in the development process at hardware level
<Name> – Activity in the development process at software level
<Name> – Artifact obtained as a result of an activity carried out in the respective development process

Figure 2.3.: ISO 26262 - The Structure of the Safety Requirements through the System Development Process (cf. [Int11c]).

during system design (cf. [SZ16]). For this purpose, the technical safety requirements are allocated to hardware and software. Those requirements which happen to be allocated to both hardware and software are further refined in order to yield exclusively hardware safety requirements (cf. [Int11e]).

The hardware design at architectural level and at the level of electrical schematics is carried out based on the specification of the hardware safety requirements and the technical system architecture (cf. [Int11e]). Each hardware component inherits the highest ASIL associated with the hardware safety requirement that are implemented by the hardware component. To avoid failures caused by a high level of complexity, ISO 26262 recommends that the hardware design is modular, has an appropriate level of granularity and avoids unnecessary complexity in the implementation of the hardware components and their respective interfaces (cf. [Int11e]). To support the hardware design, safety analyses such as FMEA or FTA can be carried out for the safety-related hardware components in order to determine possible causes of failures and the effects of faults (cf. [Int11e]). The hardware design is then verified for compliance and completeness with the hardware safety requirements using analytical methods, e.g., walk-through review and inspection, as well as simulation and hardware prototyping. Once the hardware design

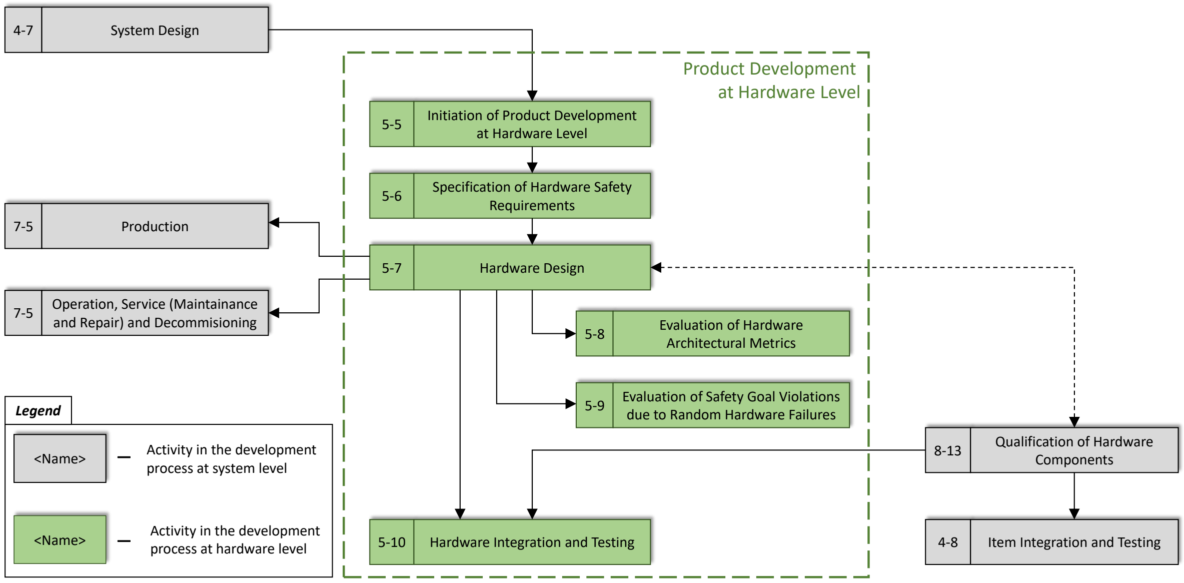

4-7
System Design
Product Development at Hardware Level
5-5
Initiation of Product Development at Hardware Level
5-6
Specification of Hardware Safety Requirements
7-5
Production
5-7
Hardware Design
7-5
Operation, Service (Maintainance and Repair) and Decommisioning
5-8
Evaluation of Hardware Architectural Metrics
5-9
Evaluation of Safety Goal Violations due to Random Hardware Failures
Legend
<Name> – Activity in the development process at system level
<Name> – Activity in the development process at hardware level
8-13
Qualification of Hardware Components
5-10
Hardware Integration and Testing
4-8
Item Integration and Testing

Figure 2.4.: ISO 26262 - The Hardware Development Process (cf. [Int11f]).

is completed, the hardware architecture of the system under development is evaluated against the requirements for fault handling as expressed by hardware architectural metrics, e.g., single point fault metric or latent fault metric (cf. [Kaf12]). The evaluation of the hardware architecture is followed by an assessment of residual risk in case of a safety goal violation due to random hardware failure (cf. [Int11e]). After the evaluation of the hardware design is completed, the developed hardware components are integrated with each other and tested. In order to derive test cases for the hardware integration test, several approaches can be applied, such as definition and analysis of equivalence classes, boundary values analysis, experience-based error guessing, analysis of environmental conditions and operational use cases, and worst-case analysis (cf. [Int11e]).

The hardware integration tests have a two-fold purpose: (1) to check the correctness and completeness of the implementation of the safety mechanisms with respect to the hardware safety requirements, e.g., by verifying the expected normal operation through functional tests and (2) to verify the robustness of the hardware against external stress stimuli, e.g., by exposing the hardware to extreme environmental conditions in worst-case tests (cf. [Int11e]).

Software Development Process in ISO 26262

The software development process according to ISO 26262 is illustrated in Figure 2.5. The process is initiated through a planing of the activities for the development at software level. These activities include the development and testing of software, as well as the coordination with the development activities at system level and at hardware level.

Before the actual software development can begin, the software safety requirements are defined based on the technical safety requirements specified at system level and the logical and the technical system architecture defined during system design. Along

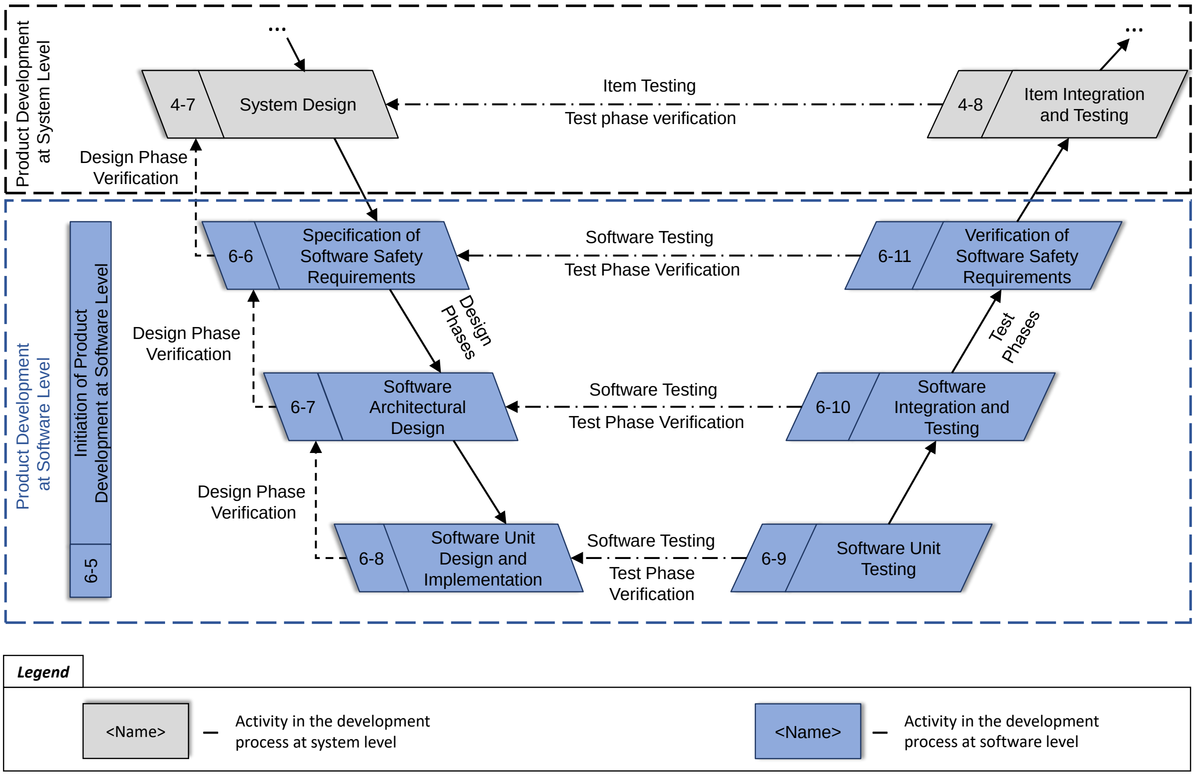

Product Development at System Level
...
4-7
System Design
Item Testing
Test phase verification
4-8
Item Integration and Testing
...
Design Phase Verification
Product Development at Software Level
Initiation of Product Development at Software Level
6-5
6-6
Specification of Software Safety Requirements
Software Testing
Test Phase Verification
6-11
Verification of Software Safety Requirements
Design Phase Verification
Design Phases
Test Phases
6-7
Software Architectural Design
Software Testing
Test Phase Verification
6-10
Software Integration and Testing
Design Phase Verification
6-8
Software Unit Design and Implementation
Software Testing
Test Phase Verification
6-9
Software Unit Testing
Legend
<Name> – Activity in the development process at system level
<Name> – Activity in the development process at software level

Figure 2.5.: ISO 26262 - The Software Development Process (cf. [Int11f]).

with these inputs from the system design, the constraints imposed by the hardware on the software are also taken into consideration in the specification of the software safety requirements (cf. [Int11f]).

The design of the software architecture is the next phase after the specification of the software safety requirements. The software architecture describes both static aspects and dynamic aspects of the software components. The static aspects of the software architecture include the logical sequence of data processing along with the data types and their characteristics, the interfaces of the software components and the external interfaces of the software, as well as the hierarchy levels in which the software components are organized (cf. [Int11f]). With respect to the dynamic aspects of the design, the software architecture describes the behavior of the software components, the data flow and the control flow between the components, the concurrency between the processes and their timing behavior, as well as the data flow at the external interfaces of the software (cf. [Int11f]). The software safety requirements formulated in the previous phase are allocated to the software components defined in the software architecture. Each software component is developed in compliance with the highest ASIL associated to any of the safety requirements allocated to it. Safety analyses, e.g., FMEA or FTA, are carried out on the software architecture in order to identify the safety-related parts of the software and the hazards that may be introduced by the respective software components. The safety analyses carried out at software architectural level also support the specification of safety mechanisms for the avoidance, reduction or mitigation of the hazards identified at software level (cf. [Int11f]). The software architecture is then checked for conformity with the established design guidelines and principles in walk-through and inspection reviews. Two examples of principles applicable to architectural design are high cohesion within each software component and restricted coupling between software components. Furthermore, the software architecture is verified for compliance with the software safety requirements through a variety of methods, e.g., static analysis, simulation, and formal verification (cf. [Int11f]). Dynamic parts of the software architecture can be verified through simulation, if executable models have been created for these parts. Formal verification can be used to provide proof of correctness with respect to the software safety requirements for critical software components to which higher ASILs are allocated, e.g., ASIL C or ASIL D (cf. [Int11f]).

Based on the software architectural design, the software units are implemented either as a model or directly as source code. The implementation is statically verified to check whether it conforms to the approved implementation conventions and standards, e.g., through static code analysis or through code reviews. Static code analysis is often provided by modern compilers and allows the software unit implementation to be examined for compliance with conventions and standards using, e.g., data flow and control flow analyses, in order to detect specific anomalies in the implementation of the software unit (cf. [SLS14b]). In addition, the software unit implementation is checked for conformance with the software architectural design through, e.g., through walk-through reviews and code inspection reviews before proceeding with the software unit testing (cf. [Int11f]).

Software unit testing is the first test phase after the software unit implementation. The purpose of this phase is to check the software units individually against their design specification. For software unit testing, various methods can be applied, such as requirements-based tests and interface tests. The appropriate test cases can be derived by analysis of the software requirements at software unit level, generation and analysis of equivalence classes, or boundary values analysis (cf. [Int11f]). There are various metrics recommended by the ISO 26262 standard in order to determine the adequacy of the derived test cases, e.g., branch coverage and modified condition/decision coverage (MC/DC) as well as coverage of the requirements (cf. [Int11f]).

Once they have passed their respective tests, the software units are integrated to larger software components, which in turn are integrated into subsystems, that eventually are put together to build the entire system under development. At each integration level, the more complex software elements obtained as a result of integration and their respective interfaces are tested against the software architectural design. Specific software integration tests are carried out with different objectives in mind. Thus, back-to-back tests can be used to verify the compliance of the integrated software with the software architecture, provided that there are executable models of the software architecture available. Verification of the hardware-software interface is also part of the software integration test and this can be done through interface tests. Requirements-based tests and fault injection tests can be used to check if the integrated software implemented the required functionality correctly. Moreover, fault injection tests can be used also to check the robustness of the integrated software. Furthermore, resource usage tests can be carried out to check that there are enough hardware resources available for the integrated software to carry out its required functionality. Analysis of requirements, generation and analysis of equivalence classes along with boundary values analysis are several methods by which test cases can be derived for the tests performed at different integration levels (cf. [Int11f]). Depending on the hierarchy level at which the integration is performed, the integrated software can be executed in specific test environments. Such tests are called *X-in-the-loop* tests, where *X* can be a model, a piece of software, a target processor or a target hardware (cf. [Int11f]). During model-in-the-loop tests, a computer model containing a representation of the test object, e.g., of a software component, as well as of its environment is run in a simulation on a host computer. Software-in-the-loop tests connect the implemented source code of the test object and a model of its environment in one simulated model. With processor-in- the-loop tests, the production source code of the test object is run on the real-time platform, while the environment of the test object is simulated separately usually on another computer. The real-time platform on which the test object is executed is connected directly to the computer running the environment simulation. In contrast to processor-in-the-loop tests, in hardware-in- the-loop tests the real-time platform running the test object is connected to a real-time simulation of its environment executed on specialized hardware, or a combination of real-time simulation and further hardware components connected electronically (cf. [KDJ$^+$16]). The adequacy of the derived test cases can be determined through specific metrics, e.g., function coverage or call coverage (cf. [Int11f]).

The final test phase at software level is the verification of the software safety requirements. The purpose of this phase is to demonstrate that the embedded software satisfies its requirements in the target environment. The verification of the software safety environments can be conducted in hardware-in-the-loop environment or in a lab environment where the system under development is partially or fully integrated, e.g., lab cars. The results of this test phase are evaluated with respect to the coverage of the software system requirements and with respect to the expected results as well as other predefined pass/fail criteria (cf. [Int11f]).

2.2.2. System Development Process according to ISO/PAS 21448

ISO 26262 introduces the concept of functional safety for E/E systems as well as their respective software in road vehicles, and provides methods to ensure functional safety in the event of system failures (cf. Section 2.2.1). However, even in the absence of system failures, the system under analysis may be unable to work as required. This is possible due to the uncertainty in the system environment which manifests itself through the appearance of unpredicted events in the environment. These events can appear through the interaction of the developed system with other systems in the environment or through the misuse of the developed system through its users. The unpredicted events which may occur in the system environment can account for edge cases, that may lead to safety hazards. These hazards do not appear as a consequence of system faults, but instead can expose functional insufficiencies of the developed system, because the system was not designed to handle the respective edge cases. In order to deal with system malfunctions in the absence of system faults, ISO/PAS 21448 introduces the notion of SOTIF (cf. Definition 2.1.14) and provides design, verification and validation methods needed to achieve SOTIF (cf. [Int19]).

The functional and system specification of the system under analysis defines relevant use cases for the system. Definition 2.2.4 introduces the notion of use case. Use cases consist of one or more scenarios, while a scenario is made up of a temporal sequence of several scenes. The concepts of scenario and scene are introduced in the Definition 2.2.5 and respectively in Definition 2.2.6.

Definition 2.2.4 - Use Case
A *use case* is a specification which belongs to a specific field of application and describes for a system under development at least one scenario in which the system is required to operate as well as its functional range, the desired behavior and the system boundaries (cf. [Int19], [UMR+15]). ■

Definition 2.2.5 - Scenario
A *scenario* is a temporal sequence of scenes and actions or events which allow the progression from one scene to another scene over a certain time span. Besides scenes and actions, a scenario can also specify the goals and values of the system under development (cf. [Int19], [UMR+15]). ■

Definition 2.2.6 - Scene
A *scene* is defined as a snapshot of the physical environment and includes dynamic elements, scenery elements, self-representation of actors and observers, as well as the relationships between these entities. An example of a dynamic element is a moving obstacle. The scenery contains information about the infrastructure in the physical environment, e.g, traffic signs and positions of traffic lights, geometry of the physical environment, e.g., number of lanes and vertical elevation, as well as environment conditions, e.g., rain or fog. The self-representation of actors and observers describes their abilities, their state and their attributes, e.g., the field of view of the system under development (cf. [Int19], [UMR+15]). ■

The use cases defined in the functional and system specification consist of scenarios, in which triggering events could occur that lead possibly to a hazardous event (cf. [Int19]). ISO/PAS 21448 differentiates between four categories of scenarios, which are depicted in Figure 2.6: (1) area A - known safe scenarios, (2) area B - known unsafe scenarios, (3) area C - unknown unsafe scenarios, and (4) area D - unknown safe scenarios.

In order to achieve SOTIF, ISO/PAS 21448 outlines the system development process and the methods used to ensure that the likelihood of a hazardous event is sufficiently low. The ISO/PAS 21448 also aims to reduce up to an acceptable level the residual risk which remains from the system not being able to process an encountered scenario in a safe manner or from the involved system users not being able to control and mitigate the hazardous event (cf. [Int19]). When a system development process is initiated according to ISO/PAS 21448, areas B and C in Figure 2.6 may be too large, which means that the

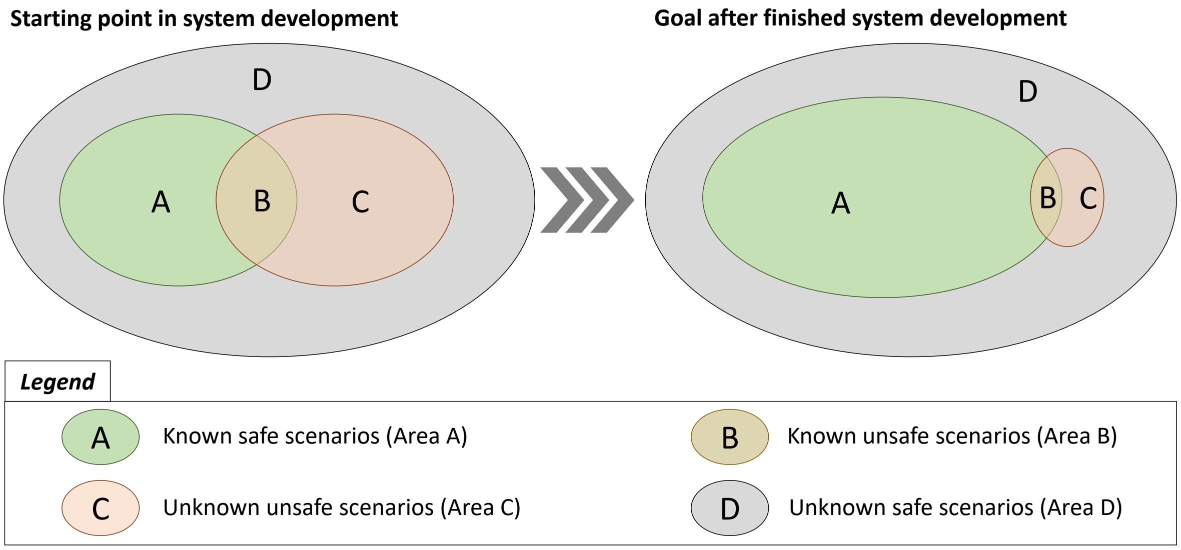

Starting point in system development
Goal after finished system development
D
A
B
C
D
A
B
C
Legend
A Known safe scenarios (Area A)
B Known unsafe scenarios (Area B)
C Unknown unsafe scenarios (Area C)
D Unknown safe scenarios (Area D)

Figure 2.6.: ISO/PAS 21448 - Visual Intuition of the Goal in the SOTIF System Development Process (cf. [Int19]).

residual risk is unacceptable. The goal of the system development process carried out according to ISO/PAS 21448 is to maximize area A, while minimizing the areas B and C (cf. [Int19]).

Figure 2.7 shows the flow of activities in the SOTIF system development process. The process starts with the functional and system specification phase. The first purpose of this phase is to define the goals of the intended functionality and the uses cases in which the intended functionality is activated, and respectively deactivated. From a functional point of view, this phase also defines the dependencies and the interaction of the system with the surrounding environment as well as the relevant environmental conditions (cf. [Int19]). For these reasons, in this thesis it is considered that the functional and system specification phase of the SOTIF development process defines the operational design domain of the system under development. The concept of operational design domain as it is understood and used in this thesis is introduced in Definition 2.2.7.

Definition 2.2.7 - Operational Design Domain
An *operational design domain (ODD)* is defined as the sum of operating conditions under which a given automated driving system is specifically designed to function, including, but not limited to, environmental, geographical, and time-of-day restrictions, as well as the presence or absence of certain traffic or roadway characteristics (cf. [SAE18]). ■

Definition 2.2.7 states that the ODD of the system under development consists of all operating conditions under which the system is designed to function. Test engineers can design specific test cases in order to show that the system works according to its requirements under the ODD conditions. Therefore, in this thesis it is considered that the ODD consists of known safe scenarios, which corresponds to area A in Figure 2.6. For this reason, in this thesis an increase of area A through the SOTIF development process is considered equivalent to an expansion of the ODD of the system under development.

The second purpose of the functional and system specification phase is to define the system architecture. The architecture includes the system components that implement the intended functionality as well as the components responsible for countermeasures in case a performance limitation of the system is exposed through an unpredicted event in the environment. These countermeasures may result in a graceful degradation of the system functionality or a warning signal for the user to take over control over the system (cf. [Int19]).

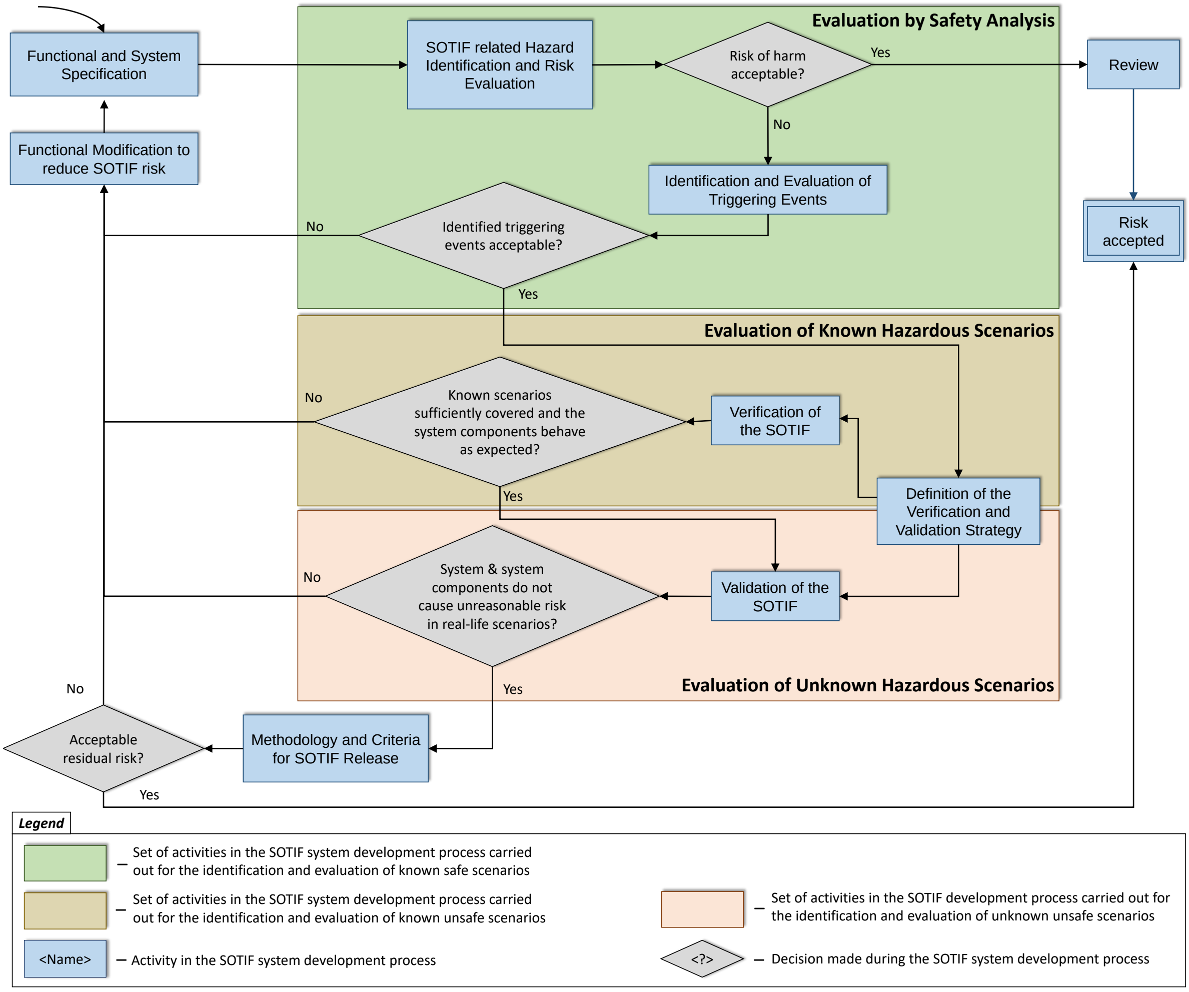

Evaluation by Safety Analysis
Functional and System Specification
SOTIF related Hazard Identification and Risk Evaluation
Risk of harm acceptable?
Yes
Review
No
Functional Modification to reduce SOTIF risk
Identification and Evaluation of Triggering Events
No
Identified triggering events acceptable?
Risk accepted
Yes
Evaluation of Known Hazardous Scenarios
No
Known scenarios sufficiently covered and the system components behave as expected?
Verification of the SOTIF
Yes
Definition of the Verification and Validation Strategy
No
System & system components do not cause unreasonable risk in real-life scenarios?
Validation of the SOTIF
No
Yes
Evaluation of Unknown Hazardous Scenarios
Acceptable residual risk?
Methodology and Criteria for SOTIF Release
Yes
Legend
– Set of activities in the SOTIF system development process carried out for the identification and evaluation of known safe scenarios
– Set of activities in the SOTIF system development process carried out for the identification and evaluation of known unsafe scenarios
– Set of activities in the SOTIF development process carried out for the identification and evaluation of unknown unsafe scenarios
<Name> – Activity in the SOTIF system development process
<?> – Decision made during the SOTIF system development process

Figure 2.7.: ISO/PAS 21448 - Operative Flowchart of the SOTIF System Development Process (cf. [Int19]).

It is worth noting that the graceful degradation of functionality is equivalent to the system under development working in a degraded operation mode, which is a suboptimal functional operation mode of the system (cf. [CPS$^+$18]). After it underwent a graceful degradation of its functionality, the system under development no longer works in the original ODD, but instead it operates in a restricted operational domain (ROD). In contrast to its ODD, which specifies all the conditions under which the system is designed to operate, the ROD of an automated driving system comprises the specific conditions under which the system is currently able to operate, which includes also the system's driving modes (cf. [CPS$^+$18]).

Once the functional and system specification is completed, the hazards that may be caused by the intended functionality are identified and the associated risks are evaluated. The same methods introduced in the concept phase of ISO 26262 system development process can be used to carry out the hazard analysis for the SOTIF related hazards, e.g., FMEA. Although the harm and controllability of the hazardous events can be estimated using the methods introduced in [Int11c], they are evaluated with respect to a specific SOTIF related hazard (cf. [Int19]). Thus, delays in the reaction time of the system user impact the evaluation of the controllability factor and are part of the SOTIF related hazard analysis. The risk evaluation considers the performance limitations of the intended functionality in order to assess whether controllability and severity are at an acceptable level. The severity and controllability evaluation take into account the expected system limitations and the countermeasures implemented to mitigate their effects (cf. [Int19]). The hazard analysis also specifies a validation target for the risk evaluation, which depends on the method chosen for the validation strategy. If deductive analysis is used, then a list of all known an relevant triggering events is considered and the validation target requires the coverage of all events on the list. In case an inductive analysis is carried out, then previously unknown triggering events are identified and the validation target requires that the triggering events do not impose unreasonable risk with a predefined statistical confidence level.

If the risk of harm is situated within acceptable bounds, then the results of the hazard analysis go through a review phase by the experts and the respective risk is considered and documented as accepted (cf. [Int19]). However, if the risks resulting from the hazard analysis are not acceptable, then the triggering events of those risks are identified and evaluated. The analysis of the triggering events takes into consideration the limitations of the system components as well as the environment conditions and foreseeable misuse of the system by its users which could expose these system limitations and create scenarios that may result in hazardous events (cf. [Int19]).

The triggering events can expose performance limitations in different system components which implement decision algorithms or in the system sensors and actuators. Consider as an example system an automated vehicle, in which an object detection algorithm rejects a person on a skateboard as a pedestrian due to its implausible speed or the camera sensor mounted in the vehicle whose performance is affected by the light in front of the vehicle (cf. [Int19]). The triggering events identified in this phase are evaluated on the acceptance criteria defined during hazard identification and risk evaluation. It is considered that the response of the system to the triggering events is acceptable with respect to SOTIF if the probability of a hazardous event caused by the system is lower than the validation target defined during hazard analysis and if there is no systematically unacceptable scenario with respect to a specific vehicle that may lead to a hazardous event (cf. [Int19]). Notice that a scenario is considered to be unacceptable if there is a high probability for a hazardous event to occur with respect to a specific vehicle, even though with respect to a entire fleet of vehicles, the probability of occurrence for the hazardous event is very low (cf. [Int19]).

If the response of the system to the identified triggering events is not considered acceptable with respect to SOTIF, the functional and system specification is updated with functional measures which improve the system design in order to reduce the risks associated with these events, but also to reduce and to mitigate SOTIF risks during system operation. Depending on the identified SOTIF risks, these measures can be aimed at risk avoidance, risk reduction or risk mitigation (cf. [Int19]). The measures can be applied to different system components, e.g., sensors, actuators, or decision algorithms. One example of a functional measure used at sensor level to improve the system and avoid SOTIF related risks is a strategy for using the appropriate sensors in order to recognize whether the system has exited its operational design domain and has encountered a known unsupported environmental condition. Another functional measure used this time at decision algorithm level to reduce the SOTIF related risks is a design strategy which incorporates warning signals and degradation of the functionality in order to handle a known unsupported SOTIF scenario (cf. [Int19]).

Mitigation measures are used in combination with mechanisms for risk avoidance and reduction. These measures can apply progressive restrictions with respect to the intended functionality: (1) restriction of the intended functionality itself, (2) restriction of the authority of specific system components with respect to the intended functionality, or (3) restriction of the overall authority of the system with respect to the intended functionality (cf. [Int19]). Consider the example of an automated vehicle equipped with a lane keeping assist system. In case the lane is not detected accordingly, the lane keeping assist functionality is restricted in order to avoid undesired steering. In case the performance of a camera sensor mounted on the automated vehicle is affected by the reflection of the surrounding light, then the camera sensor is disabled and the operation of the automated driving function continues with restricted authority, using radar and other sensors. In case all sensors of the automated vehicle are incapacitated in a snow storm, then the automated vehicle hands over the control over the driving function to the vehicle driver (cf. [Int19]). If the identified risks are related to the foreseeable misuse of the system by its users, then measures that prevent the user from inadvertent use of the system or monitoring systems that warn the user in case an incorrect system operation is detected are added to the functional and system specification. An example of the latter is a monitoring system which warns the driver of an automated vehicle when the steering wheel has been released. If the risk presented by the triggering events is considered acceptable, the system development process proceeds with the definition of the verification and validation strategy.

The verification and validation strategy is defined with respect to the SOTIF rationale and it develops the necessary procedures, which are used to demonstrate SOTIF for the system under analysis (cf. [Int19]). The verification and validation strategy takes the following artifacts as inputs to define its procedures: functional concept of the system, system architecture, results of the hazard analysis and risk assessment, the system integration and testing plan (cf. [Int19]). The strategy focuses on two of the four areas depicted in Figure 2.6: (1) area B - known unsafe scenarios and (2) area C - unknown unsafe scenarios.

Verification of SOTIF focuses on area B in Figure 2.6 and its objective is to verify the system and its components - sensors, actuators, and decision algorithms - to show that these behave according to their specification with respect to known hazardous events and foreseeable misuse. This phase of the system development process takes into account the defined verification strategy, the system's functional concept, the verification targets, as well as the results of the hazard analysis and risk assessment (cf. [Int19]). The verification strategy comprises methods which demonstrate the functional performance of system sensors, the ability of decision algorithms to react as required and avoid unwanted actions, as well as the performance of the system actuators in case of intended use and in case of foreseeable misuse. Furthermore, the verification strategy includes methods to check the robustness and controllability of the integrated system (cf. [Int19]). If under given specific hazardous events, there are still system components which do no fulfill their specification or the developed test cases do not provide enough test coverage for the system and its components, then further functional measures are added to the functional and system specification. These measures may restrict the system functionality in such a way so that the respective system components and the system itself fulfill their requirements with respect to the identified hazardous events. However, if the known hazardous events are sufficiently covered by test cases and the system components and the entire system comply with their specification, then the system development process proceeds with the validation of SOTIF.

The objective of the SOTIF validation phase is to show that the system and its components - sensors, actuators, decision algorithms - do not cause any unreasonable risk in real-life use cases. The focus of the SOTIF validation is area C in Figure 2.6, which covers unknown hazardous scenarios that may occur in real-life use cases. This phase of the system development process takes as inputs the validation strategy, the verification results obtained in the previous phase for the defined used cases, the system's functional concept, the validation targets, and the results of the hazard analysis and risk assessment (cf. [Int19]). In order to show that the system does not cause any unreasonable risks in real-life scenarios, the validation strategy must provide evidence that the system meets the validation targets defined during hazard analysis. If the system does not meet its validation targets, further measures are added to the functional and system specification, which modify the system design so that the system is able to meet its validation targets (cf. [Int19]).

Nevertheless, if the system is able to fulfill its validation targets, then the methodology for the evaluation of SOTIF for release is developed. The purpose of this methodology is to review the SOTIF activities carried out during the system development process, and based on the results of these activities, evaluate whether the residual risk is acceptable or not (cf. [Int19]). During the evaluation for the SOTIF release, the verification and validation process is subject to scrutiny and it is checked whether all use cases formulated with respect to the intended functionality have been taken account during validation of SOTIF and whether the defined test cases have covered all the triggering events identified in the safety analysis (cf. [Int19]). Furthermore, it is checked whether the intended functionality has been exercised sufficiently in order to evaluate both nominal and potential unwanted behavior. In case unintended behavior was observed which potentially leads to a hazardous event, the evaluation for the SOTIF release checks whether enough evidence has been provided in order to argue the absence of unreasonable risk (cf. [Int19]).

2.3. Property Specification for Automated Safety-critical Systems

The system development process according to ISO 26262 and ISO/PAS 21448 has been presented in Section 2.2.1 and respectively Section 2.2.2. Testing is the principal verification method recommended by these standards to show that a system under development behaves according to its system requirements (cf. [Int11f], [Int11a], [Int19]). Formal verification is another method recommended by ISO 26262 for the verification of system components and automotive functions with higher ASIL, e.g. ASIL C or ASIL D, against their functional and safety requirements (cf. [Int11f]). Formal verification methods need a formal properties specification as well as a formal model of the system or the system component which is to be verified. The formal properties specification is derived from the system requirements and the safety requirements derived during the item definition and the safety analysis in the concept phase. Formalisms used for modeling automated safety-critical systems are presented in Section 2.4, while this section is concerned with formalisms and formal languages that used for the formal specification of requirements in automated safety-critical systems.

Luckuck et al. carry out a survey on the formal specification and verification for autonomous robotic systems (cf. [LFD$^+$19]) and uncover a wide range of formalisms used for property specification for robotic systems, e.g., different variants of temporal logic (cf. [WFCJ11], [DFSW16], [DWFZ12], [**?**], [HEZ$^+$14], [IQV16], [MV09]), dynamic logic (cf. [SKA13]), and process algebra (cf. [OAH$^+$14]). In an effort to define the suitable selection criteria of a formal method for an industrial application, Kossak and Mashkoor discover formalisms which are used in various industrial domains for the formal specification of automated safety-critical systems (cf. [KM16]), e.g., higher order logic combined with probabilistic analysis for machine control systems (cf. [MHB13]), Event-B for transportation and platooning systems (cf. [MJ11], [MJS], [Lan08]).

For automated safety-critical systems developed according to the standards ISO 26262 or ISO/PAS 21448 it is important to ensure their functional safety and respectively the safety of their intended functionality. This means that any verification procedure used during the system development process must provide evidence that the system under development is free of unreasonable risks, i.e. the probability of occurrence for the hazardous events associated with the respective risks is low enough so that the residual risk remains at an acceptable level. Making sure that specific hazardous events do not occur at all or their probability of occurrence is below an accepted threshold can be expressed using safety properties. At the same time, it is important for an automated safety-critical system to show that it progresses with the realization of its tasks and achieves its predefined goals. This can be expressed through liveness properties. The notions of safety property and liveness property are introduced by Lamport in [Lam77], and are reiterated in this thesis in Definitions 2.3.1 and 2.3.2.

Definition 2.3.1 - Safety Property
A safety property is defined as a property which states that a certain event will not happen or a certain condition will not occur (cf. [Lam77]). ■

Definition 2.3.2 - Liveness Property
A liveness property is a property which requires that an event must happen or a condition must occur eventually (cf. [Lam77]). ■

This section introduces the two temporal logics used in this thesis to express properties for automated safety-critical systems: timed computational-tree logic in Section 2.3.1 and probabilistic computational tree logic in Section 2.3.2. This includes the syntax and the semantics of the respective temporal logic.

2.3.1. Timed Computation Tree Logic

The basis of timed computation tree logic (TCTL) is formed by computation tree logic (CTL), which is introduced by Clarke and Emerson in [CE81] and respectively in [EC82]. Timed CTL is an extension of CTL, which is defined by Alur et al. in [ACD90], in order to express properties of real-time systems (cf. [BK08]). Timed CTL formulae are formulated over the atomic propositions and the clock constraints, which are specific to the timed automata modeling the system under analysis. The concept of timed automata and details related to it are discussed in Section 2.4.1.

Before introducing the syntax of TCTL, the notions of clock interpretation and clock constraint, as well as the satisfaction relation for clock constraints must be defined. This is done in Definitions 2.3.3 and Definition 2.3.3. The TCTL syntax and its semantics are presented in Definition 2.3.6 and Definition 2.3.7.

Definition 2.3.3 - Clock Interpretation
Let C be a set of real-valued variables called clocks. A *clock interpretation* for the set C is a mapping $\nu : C \to \mathbb{R}_{\geq 0}$ from C to the set of non-negative real numbers $\mathbb{R}_{\geq 0}$ which assigns a real value to each clock in C (cf. [AH99]). ■

Notation 2.3.1. $\nu + \delta$ denotes the clock interpretation which increases the value of every clock $x \in C$ by $\delta \in \mathbb{R}$. For a set of clocks $D \subseteq C$, $\nu[D := 0]$ denotes the clock interpretation which resets every clock in $x \in D$ and maintains all the other clocks $y \in C \setminus D$ unchanged (cf. [AH99]). The notion of clock interpretation given in Definition 2.3.3 is equivalent to the notion of clock valuation introduced by Baier and Katoen in [BK08]. ■

Definition 2.3.4 - Clock Constraint
A *clock constraint* over a set of clocks C is either an atomic clock constraint or a conjunction of atomic clock constraints.

An *atomic clock constraint* compares a clock value with a time constant taken from the set of non-negative rationals $\mathbb{Q}_{\geq 0}$. An atomic clock constraint does not contain any conjunctions. The following grammar defines the syntax of clock constraints:

$$\phi_{\mathit{Clk}} ::= x \leq c \mid x < c \mid x \geq c \mid x > c \mid \phi^1_{\mathit{Clk}} \wedge \phi^2_{\mathit{Clk}}$$

where $x \in C$ is a clock variable and $c \in \mathbb{N}$ is a constant (cf. [AH99], [BK08]). ■

Notation 2.3.2. $\mathcal{B}(C)$ denotes the set of clock constraints over the set C (cf. [BY04]). ■

Definition 2.3.5 - Satisfaction Relation for Clock Constraints
Let C be a set of clocks, $x \in C$ a clock, ν a clock interpretation, ϕ^1_{Clk} and ϕ^2_{Clk} two clock constraints, and $c \in \mathbb{N}$ a constant. Then, the satisfaction relation $\models$ is defined for clock constraints as follows (cf. [BK08]):

$$\begin{array}{lll}
\nu \models \mathit{true} & & \\
\nu \models x < c & \mathit{iff} & \nu(x) < c \\
\nu \models x \leq c & \mathit{iff} & \nu(x) \leq c \\
\nu \models \neg\phi^1_{\mathit{Clk}} & \mathit{iff} & \nu \not\models \phi^1_{\mathit{Clk}} \\
\nu \models \phi^1_{\mathit{Clk}} \wedge \phi^2_{\mathit{Clk}} & \mathit{iff} & \nu \models \phi^1_{\mathit{Clk}} \ \wedge \ \nu \models \phi^2_{\mathit{Clk}}
\end{array}$$

■

Definition 2.3.6 - Syntax of TCTL
The formulae of TCTL are either state or path formulae. The *state formulae* over the set of atomic propositions AP and the set of clocks C are built according to the following grammar (cf. [BK08]):

$$\psi_{\mathit{State}} ::= \mathit{true} \mid a \mid \phi_{\mathit{Clk}} \mid \psi^1_{\mathit{State}} \wedge \psi^2_{\mathit{State}} \mid \neg\ \psi_{\mathit{State}} \mid E\ \psi_{\mathit{Path}} \mid A\ \psi_{\mathit{Path}}$$

where ψ_{State} is a state formula, ψ_{Path} is a path formula, a is an atomic proposition, and ϕ_{Clk} is a clock constraint. The *path formulae* are defined as:

$$\psi_{\mathit{Path}} ::= \psi^1_{\mathit{State}}\ U_J\ \psi^2_{\mathit{State}}$$

where U_J denotes the *Until* operator inherited from CTL and associated with the interval $J \subset \mathbb{R}_{\geq 0}$. ■

Notation 2.3.3. The interval $J \subset \mathbb{R}_{\geq 0}$ is an interval of real numbers which has natural numbers as lower and upper bounds, i.e. the form of interval J is either $[n, m]$, (n, m), $(n, m]$, or $[n, m)$, with $n, m \in \mathbb{N}$ and $n \leq m$ (cf. [BK08]). ■

A property expressed in TCTL is always a state formula. Path formulae appear only together with path operators A (*for All Paths*) and E (*for Some Path*).

The other propositional logic operators, such as $\vee$ (logical *or*) and $\rightarrow$ (logical *implication*), are derived from $\wedge$ (logical *and*) and $\neg$ (logical *negation*) using the laws of Boolean algebra, e.g., De Morgan laws and the material implication rule. The operators A and E are inherited from CTL and they only quantify over time divergent paths in a given timed automaton.

Notation 2.3.4. Let TA be a timed automaton as in Definition 2.4.1, $\mathit{TS}(\mathit{TA})$ the corresponding transition system as in Definition 2.4.2, and (s, ν) a state in $\mathit{TS}(\mathit{TA})$. A path of the form $\pi = (s_0, \nu_0) \xrightarrow{\delta_0} \xrightarrow{a_1} (s_1, \nu_1) \xrightarrow{\delta_1} \xrightarrow{a_2} (s_2, \nu_2) \xrightarrow{\delta_2} \xrightarrow{a_3} \ldots$ is time divergent if time progresses always on this path, i.e. $\sum_i \delta_i$ diverges. The set $\mathit{Path}_{\mathit{div}}((s, \nu))$ denotes the set of time-divergent paths that start in the state (s, ν). ■

Intuitively, the formula $A\ \psi_{\mathit{Path}}$ requires that the path formula ψ_{Path} holds on all paths of the timed automaton that models the system under development. In contrast, the formula $E\ \psi_{\mathit{Path}}$ stipulates that there exists a path in the given timed automaton on which the path formula ψ_{Path} holds (cf. [BK08]). The modal operators G (*Globally*) and F (*Eventually*) are also imported from CTL and are used to quantify over states within a path. The operator G requires that all states in the respective path satisfy a given property, while the operator F asks that there is at least one state in the execution path which fulfills the respective property (cf. [BY04]). These operators have their timed variants, which are expressed with the help of the U_J (timed *Until*) operator (cf. [BK08]):

$$F_J\ \psi_{\mathit{State}} = \mathit{true}\ U_J\ \psi_{\mathit{State}} \quad EG_J\ \psi_{\mathit{State}} = \neg AF_J\ \neg\psi_{\mathit{State}} \quad AG_J\ \psi_{\mathit{State}} = \neg EF_J\ \neg\psi_{\mathit{State}}$$

There are CTL operators which are not represented in TCTL, i.e. the X (*Next*) operator is absent in TCTL. Since time is considered to be continuous in TCTL, there is no unique next time instant that can be meaningfully represented through an X (*Next*) operator (cf. [BK08]).

When asking whether a timed automaton TA satisfies a TCTL property, the respective TCTL state formula is interpreted over the transition system $\mathit{TS}(\mathit{TA})$ which describes the formal operational semantics of the timed automaton TA. The notion of timed automata as its operational semantics are introduced in Section 2.4.1. It is said that a timed automaton TA satisfies a TCTL state formula ψ_{State}, denoted $TA \models \psi_{\mathit{State}}$ if and only if ψ_{State} is satisfied in all initial states (s_0, ν_0) of the associated transition system $\mathit{TS}(\mathit{TA})$, denoted as $(s_0, \nu_0) \models \psi_{\mathit{State}}$. The satisfaction relation $\models$ is introduced in Definition 2.3.7.

Definition 2.3.7 - Semantics of TCTL

Let $\mathit{TA} = (\mathit{Loc}, \mathit{Loc}_{\mathit{Init}}, \mathit{Act}, C, \mathit{Inv}, E, AP, L)$ be a timed automaton as defined in Definition 2.4.1. Let $a \in AP$ be an atomic proposition, $\phi_{\mathit{Clk}} \in \mathcal{B}(C)$ an atomic clock constraint, and $J \subseteq \mathbb{R}_{\geq 0}$ an interval of real numbers. Let $\mathit{TS}(\mathit{TA})$ be the transition system associated with the timed automaton TA as per Definition 2.4.2 and (s, ν) be a state in $\mathit{TS}(\mathit{TA})$. Let ψ^1_{State} and ψ^2_{State} be two TCTL state formulae and ψ_{Path} a TCTL path formula. Then, the satisfaction relation $\models$ is defined for state formulae by (cf. [BK08]):

$$\begin{array}{lll}
(s,\nu) \models \mathit{true} & & \\
(s,\nu) \models a & \mathit{iff} & a \in L(s) \\
(s,\nu) \models \phi_{\mathit{Clk}} & \mathit{iff} & \nu \models \phi_{\mathit{Clk}} \\
(s,\nu) \models \neg\psi^1_{\mathit{State}} & \mathit{iff} & \mathit{not}\ (s,\nu) \models \psi^1_{\mathit{State}} \\
(s,\nu) \models \psi^1_{\mathit{State}} \wedge \psi^2_{\mathit{State}} & \mathit{iff} & (s,\nu) \models \psi^1_{\mathit{State}}\ \wedge\ (s,\nu) \models \psi^2_{\mathit{State}} \\
(s,\nu) \models E\ \psi_{\mathit{Path}} & \mathit{iff} & \exists \pi \in \mathit{Path}_{\mathit{div}}((s,\nu)),\ \mathit{s.t.}\ \pi \models \psi_{\mathit{Path}} \\
(s,\nu) \models A\ \psi_{\mathit{Path}} & \mathit{iff} & \forall \pi \in \mathit{Path}_{\mathit{div}}((s,\nu)),\ \pi \models \psi_{\mathit{Path}}
\end{array}$$

Given the time-divergent path $\pi = (s_0, \nu_0) \xrightarrow{\delta_0} \xrightarrow{a_1} (s_1, \nu_1) \xrightarrow{\delta_1} \xrightarrow{a_2} (s_2, \nu_2) \xrightarrow{\delta_2} \xrightarrow{a_3} \dots$, $\alpha_i \in \mathit{Act}$ and the path formula $\psi^1_{\mathit{State}}\ U_J\ \psi^2_{\mathit{State}}$, then $\pi \models \psi^1_{\mathit{State}}\ U_J\ \psi^2_{\mathit{State}}$ if and only if there exists a state $(s_i, \nu_i + \delta)$, where $i \geq 0$ and $\delta \in [0, \delta_i]$, so that:

$$(s_i, \nu_i + \delta) \models \psi^2_{\mathit{State}} \text{ \textit{with} } \sum_{k=0}^{i-1} (\delta_k + \delta) \in J$$

and for all the previous states $(s_j, \nu_j + \delta')$, where $j \geq 0, j \leq i$ and $\delta' \in [0, \delta_j]$ the following holds (cf. [BK08]):

$$(s_j, \nu_j + \delta') \models \psi^1_{\mathit{State}} \vee \psi^2_{\mathit{State}} \text{ \textit{with} } \sum_{k=0}^{j-1} (\delta_k + \delta') \leq \sum_{k=0}^{i-1} (\delta_k + \delta)$$

■

Notice that Definition 2.3.7 does not consider only the delays δ and δ' which may occur in the states (s_i, ν_i) and respectively (s_j, ν_j). Instead it considers all the delays δ_k, that may have occurred in each previous state (s_k, ν_k) starting from the initial state (s_0, ν_0) of path π, and requires that the respective sums of these delays leading up to the states (s_j, ν_j) and respectively (s_i, ν_i) are situated in the interval of real numbers J.

Uppaal is a verification tool which accepts a fragment of TCTL as a property specification language (cf. [BY04]). Figure 2.8 gives a visual intuition of the TCTL formulae that are used in Uppaal. Notice that Uppaal uses the symbols $\Box$ and $\Diamond$ for the G operator and respectively for the F operator. Uppaal also introduces a textual notation similar to the C programming language, which allows system designers more flexibility in modeling the behavior of the system under analysis through custom defined variables and functions. This reflects also on the specification of the system properties, as these can contain not only clock constraints but also predicates over the user defined variables.

The TCTL fragment implemented in Uppaal allows the system designer to express three categories of system properties: reachability properties, safety properties, and liveness properties. Safety properties are formulated as one of either of the two forms: $A\Box\phi$ or $E\Box\phi$. The safety property $A\Box\phi$ expresses that ϕ is on any path always true, while $E\Box\phi$ says that there exists a path on which ϕ is always true. Liveness properties in Uppaal can express that either a condition ϕ is eventually satisfied using the formula $A\Diamond\phi$, or that always the occurrence of the event ϕ leads to the condition ψ being eventually satisfied, by using the formula $\phi \leadsto \psi$ (cf. [BDL04]).

2.3.2. Probabilistic Computational Tree Logic

Probabilistic computational tree logic (PCTL) was introduced in [HJ94], [BdA95], and [BK98] as probabilistic variant of CTL, in order to express properties of probabilistic systems with and without nondeterministic behavior. Instead of properties that require that a certain condition is satisfied on all execution paths or just some paths of a system,

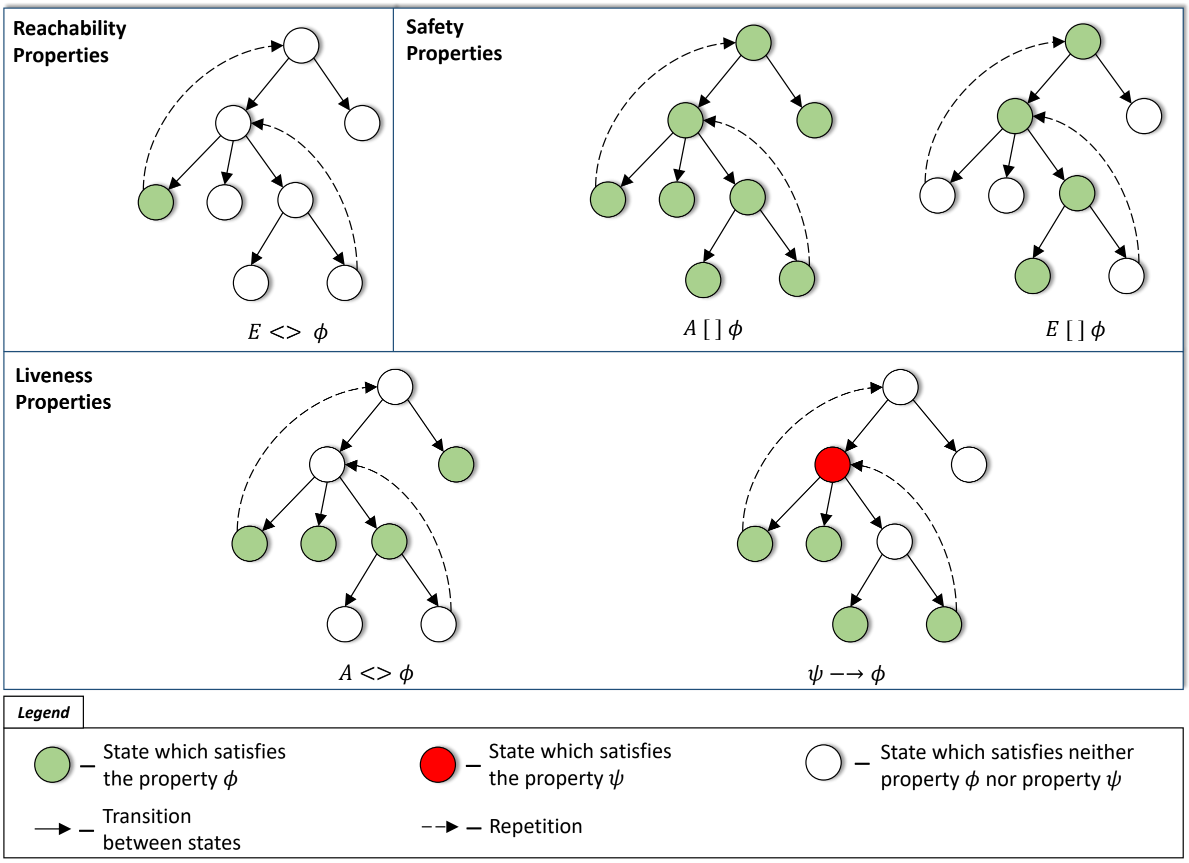

Reachability Properties
E <> ϕ
Safety Properties
A [] ϕ
E [] ϕ
Liveness Properties
A <> ϕ
ψ −→ ϕ
Legend
– State which satisfies the property ϕ
– State which satisfies the property ψ
– State which satisfies neither property ϕ nor property ψ
– Transition between states
– Repetition

Figure 2.8.: Visual Intuition of TCTL Formulae used in UPPAAL (cf. [BDL04]).

PCTL allows the specification of properties which impose constraints on the proportion of the system execution paths that satisfy the respective condition (cf. [BdAFK18]).

PCTL extends the logic CTL with a probability operator $\mathbf{P}_{\bowtie\ p}$. Thus, PCTL defines the standard propositional logic operators along with the probabilistic operator $\mathbf{P}_{\bowtie\ p}(\phi)$, where ϕ is a path formula and $\bowtie\ p$ is a probability constraint (cf. [BK08], [BdAFK18]). Intuitively, the formula $\mathbf{P}_{\bowtie\ p}(\phi)$ requires that the probability of taking a path satisfying ϕ meets the probability constraint $\bowtie p$. In the probability constraint $\bowtie p$, $\bowtie$ is a comparison operator in $\{<, \leq, >, \geq\}$ and $p \in [0, 1] \cap \mathbb{Q}$ is a probability threshold (cf. [BdAFK18]). The probability operator $\mathbf{P}_{\bowtie\ p}$ is considered to be the quantitative counterpart to the universal path quantifier A and existential path quantifier E defined in CTL (cf. [BK08]). Baier et al. extend PCTL with an expectation operator, which helps reasoning about the accumulated cost and the instantaneous cost for a given path ϕ (cf. [BdAFK18]). Rather than asking about the cost associated with executing certain paths, the properties formulated for a system under analysis in this thesis are safety properties. Such properties require for example that, given the probability of occurrence for a hazardous event, the probability with which the system remains in a safe state satisfies a predefined threshold. Definition 2.3.8 introduces the syntax of PCTL as it is later used in this thesis.

Definition 2.3.8 - Syntax of PCTL
The formulae of PCTL consist of state formulae and path formulae. The *state formulae* are expressed over the set of atomic propositions AP according to the following grammar (cf. [BK08], [FKNP11]):

$$\psi_{State} ::= \mathit{true} \mid a \mid \psi^1_{State} \land \psi^2_{State} \mid \neg\ \psi_{State} \mid \mathbf{P}_{\bowtie\ p}(\psi_{Path})$$

where ψ_{State} is a state formula, ψ_{Path} is a path formula, a is an atomic proposition, $\bowtie\ \in \{<, \leq, >, \geq\}$, and $p \in [0,1] \cap \mathbb{Q}$.
The *path formulae* are defined as:

$$\psi_{Path} ::= X\ \psi_{State} \mid \psi^1_{State}\ U\ \psi^2_{State} \mid \psi^1_{State}\ U_{\leq n}\ \psi^2_{State}$$

where $U_{\leq n}$ denotes the step-bounded *Until*, with $n \in \mathbb{N}$. ■

A property expressed in PCTL is always a state formula. Path formulae are used only inside the probabilistic operator $\mathbf{P}_{\bowtie\ p}$. Similar to TCTL, the other propositional logic operators are derived using the laws of Boolean algebra. With respect to the grammar of path formulae, PCTL extends CTL by adding $U_{\leq n}$, the step-bounded *Until* operator. Intuitively, the path formula $\psi^1_{State}\ U_{\leq n}\ \psi^2_{State}$ asserts that ψ^2_{State} shall be satisfied with maximum n steps, and that ψ^1_{State} holds in all the states traversed before reaching a state in which ψ^2_{State} is satisfied (cf. [BK08]).

Besides the operators X (*Next*), U (*Until*) and $U_{\leq n}$ (bounded *Until*), PCTL allows further operators in the path formulae: operator F (*Eventually*) and G (*Globally*). The F operator is obtained by using the U operator, while the G operator is defined by the duality with the F operator:

$$F\ \psi_{State} = \mathit{true}\ U\ \psi_{State} \qquad\qquad G\ \psi_{State} = \neg(F\ \neg\psi_{State})$$

The corresponding bounded operators are obtained in a similar manner, using the $U_{\leq n}$ operator:

$$F_{\leq n}\ \psi_{State} = \mathit{true}\ U_{\leq n}\ \psi_{State} \qquad\qquad G_{\leq n}\ \psi_{State} = \neg(F_{\leq n}\ \neg\psi_{State})$$

In the context of PCTL, the duality between the G and F operators reflects also on the probability operator $\mathbf{P}_{\bowtie\ p}$. This means that an event e occurs with a probability of at most p if its complement event $e^{\complement}$ occurs with a probability of at least $1-p$ (cf. [BK08]). This allows the definition of the following relations:

$$\mathbf{P}_{\leq p}(G\ \psi_{State}) = \mathbf{P}_{\geq 1-p}(F\ \neg\psi_{State}) \qquad \mathbf{P}_{\leq p}(G_{\leq n}\ \psi_{State}) = \mathbf{P}_{\geq 1-p}(F_{\leq n}\ \neg\psi_{State})$$

In this thesis, PCTL formulae are interpreted over a Markov decision process $\mathcal{M}$. The semantics of PCTL is defined with respect to a class of schedulers $\mathit{Sched}_{\mathcal{M}}$ of a Markov decision process $\mathcal{M}$. The notion of scheduler as well as the definition and operational semantics of Markov decision processes are given in Section 2.4.3. It is said that a PCTL state formula ψ_{State} is satisfied in a state s of a Markov decision process $\mathcal{M}$, if under any scheduler $\mathcal{U} \in \mathit{Sched}_{\mathcal{M}}$, $s \models_{\mathit{Sched}_{\mathcal{M}}} \psi_{State}$ (cf. [FKNP11], [BdAFK18]). The satisfaction relation $\models_{\mathit{Sched}_{\mathcal{M}}}$ is described in Definition 2.3.9.

Definition 2.3.9 - Semantics of PCTL
Let $\mathcal{M} = (S, Act, \mathbf{P}, s_{init}, AP, L)$ be a Markov decision process as in Definition 2.4.6 and s a state in $\mathcal{M}$. Let $a \in AP$ be an atomic proposition. Let ψ^1_{State} and ψ^2_{State} be two state formulae and ψ_{Path} a path formula. Let p be a probability threshold and $Sched_{\mathcal{M}}$ a class of schedulers of $\mathcal{M}$ as in Definition 2.4.8. Then, the satisfaction relation $\models_{Sched_{\mathcal{M}}}$ is defined for state formulae by (cf. [FKNP11],[BdAFK18]):

$$\begin{array}{lll}
s \models_{Sched_{\mathcal{M}}} true & & \\
s \models_{Sched_{\mathcal{M}}} a & \textit{iff} & a \in L(s) \\
s \models_{Sched_{\mathcal{M}}} \neg\psi^1_{State} & \textit{iff} & s \not\models_{Sched_{\mathcal{M}}} \psi^1_{State} \\
s \models_{Sched_{\mathcal{M}}} \psi^1_{State} \wedge \psi^2_{State} & \textit{iff} & s \models_{Sched_{\mathcal{M}}} \psi^1_{State} \ \wedge \ s \models_{Sched_{\mathcal{M}}} \psi^2_{State} \\
s \models_{Sched_{\mathcal{M}}} \mathbf{P}_{\bowtie\ p}(\psi_{Path}) & \textit{iff} & \forall\, \mathcal{U} \in Sched_{\mathcal{M}}, Pr^{\mathcal{U}}_s(\psi_{Path}) \bowtie p
\end{array}$$

where $Pr^{\mathcal{U}}_s(\psi_{Path}) \stackrel{\text{def}}{=} Pr^{\mathcal{U}}_s(\{\pi \in Paths_{\mathcal{M}} \mid \pi \models_{Sched_{\mathcal{M}}} \psi_{Path}\})$.

For any path $\pi \in Paths_{\mathcal{M}}$, the satisfaction relation $\models_{Sched_{\mathcal{M}}}$ for path formulae is defined by (cf. [BK08],[FKNP11]):

$$\begin{array}{lll}
\pi \models_{Sched_{\mathcal{M}}} X\ \psi_{State} & \textit{iff} & \pi[1] \models_{Sched_{\mathcal{M}}} \psi_{State} \\
\pi \models_{Sched_{\mathcal{M}}} \psi^1_{State}\ U\ \psi^2_{State} & \textit{iff} & \exists\ j \geq 0\ s.t.\ \pi[j] \models_{Sched_{\mathcal{M}}} \psi^2_{State} \wedge \\
 & & \forall\ 0 \leq k < j\ s.t.\ \pi[k] \models_{Sched_{\mathcal{M}}} \psi^1_{State} \\
\pi \models_{Sched_{\mathcal{M}}} \psi^1_{State}\ U_{\leq n}\ \psi^2_{State} & \textit{iff} & \exists\ 0 \leq j \leq n\ s.t.\ \pi[j] \models_{Sched_{\mathcal{M}}} \psi^2_{State} \wedge \\
 & & \forall\ 0 \leq k < j\ s.t.\ \pi[k] \models_{Sched_{\mathcal{M}}} \psi^1_{State}
\end{array}$$

■

Notation 2.3.5. Given a Markov decision process $\mathcal{M}$, a scheduler $\mathcal{U} \in Sched_{\mathcal{M}}$, and an s in $\mathcal{M}$, then:

- $Pr^{\mathcal{U}}_s(\psi_{Path})$ denotes the probability that the path formula ψ_{Path} is satisfied by state s in $\mathcal{M}$ under the scheduler $\mathcal{U}$, and
- $Pr^{\mathcal{U}}_s(\{\pi \in Paths_{\mathcal{M}} \mid \pi \models_{Sched_{\mathcal{M}}} \psi_{Path}\})$ denotes probability that all infinite paths π starting in state s satisfy the path formula ψ_{Path} under the scheduler $\mathcal{U}$. ■

Notice that the only difference between the operators U and $U_{\leq n}$ is that the index j over the states of the path π is unbounded in the case of the former and is bounded by n in the case of the latter. Intuitively, this means that $\psi^1_{State}\ U\ \psi^2_{State}$ is true if ψ^2_{State} is satisfied at some step in the future, and ψ^1_{State} holds up until that step. In comparison, $\psi^1_{State}\ U_{\leq n}\ \psi^2_{State}$ is true if ψ^2_{State} is satisfied within n steps, and ψ^1_{State} holds up until ψ^2_{State} becomes true.

Given an Markov decision process $\mathcal{M}$ with s_{init} its initial state, it is said that $\mathcal{M}$ satisfies a PCTL state formula ψ_{State} if and only if $s_{init} \models_{Sched_{\mathcal{M}}} \psi_{State}$ (cf. [BdAFK18]).

Prism and Storm are two verification tools which accept a rich range of property specification languages, among which is also PCTL (cf. [KNP02], [DJKV17], [Hen18]).

2.4. System Modeling for Automated Safety-Critical Systems

The system architecture is created during the system design phase of the system development process presented in Section 2.2.1 and respectively Section 2.2.2. The software architecture represents all the software components of the system under analysis along with their respective interfaces and the relations between them. The software components may be modeled via graphical notations, e.g., UML 2, and then implemented manually, or the software components can be created as executable models, from which the respective source code is automatically generated with the help of a toolchain which supports model-based engineering. In industrial domains such as automotive or aeronautics, there are commercial toolchains which have established themselves over years long experience, e.g., Matlab/Simulink [BD97], Ascet-sd [LBBZ97], or Ansys Scade [CPP17].

In the survey in [LFD+19], Luckuck et al. point out a wide range of methods used to describe in the software architecture of autonomous robotic systems. Several of these methods produce executable models, from which the respective source code is generated. Some of these methods use executable graphical notations such as restricted Finite State Machines [MRW06] and ArmarX statecharts [WOK+16] to produce executable C++ code. Model-driven engineering approaches, e.g., the Brics component model [BKH+13] and the architecture description language MontiArcAutomaton [RRRW14], translate models into platform-specific component models or into source code aimed at specific robotic platforms, e.g., Ros (cf. [LFD+19]). The system requirements can then be checked through the execution of the executable software program or through the simulation of the executable models.

Nevertheless, for a highly safety-critical system function, e.g., an automotive function with ASIL C or ASIL D, irrefutable proof is necessary that the respective function satisfies its functional and safety requirements. Formal verification methods can provide mathematical proof for the correctness of safety-critical systems with respect to their specified system requirements. This is achieved based on the unambiguous semantics of the property specification which formalizes the verification goal and of the system model which formally describes the system to be verified. Luckcuck et al. [LFD+19] survey several methods used for the formal modeling of autonomous robotic systems and their environment. Many of these methods rely on state-transition models, e.g., Petri nets (cf. [CDG07], [CL07]), [DWFZ12], [FJN+11], finite transition systems ([FJN+11], [DWFZ12], [GJD13], [WDF+16]), or probabilistic models such as discrete-time Markov chains and Markov decision processes (cf. [KDF12]). Other approaches use set-based formalisms, e.g., Z specification (cf. [LDSW09]) or Event-B ([TPT+12]), dynamic logic (cf. [SKA13]), process algebra (cf. [OAH+14], [MBL+13]) or ontologies (cf. [MV09], [PCR+13]).

This section introduces two formalisms used in this thesis for the modeling automated safety-critical systems: timed automata in Section 2.4.1 and Markov decision processes in Section 2.4.3. Before the introduction of Markov decision processes in Section 2.4.3, Section 2.4.2 gives a short overview of modeling formalisms for probabilistic safety-critical systems. The discussion of timed automata and Markov decision processes in Section 2.4.1 and respectively Section 2.4.3 includes the syntax and semantics for each formalism. However, in order to reason about an automated safety-critical system in the environment in which it is designed to operate, both the system and the environment have to be modeled and their models have to be put in interaction with each other. This is done through parallel composition of the two models. Therefore, for each modeling formalism presented in this section, it is illustrated how the parallel composition is built.

2.4.1. Timed Automata

Timed automata represent a formalism introduced by Alur et al. in [AD92] and [AD94] in order to model the behavior of real-time critical systems. In 1999, Alur extended the definition of timed automata with the notion of location invariants (cf. [AH99]). A timed automaton is essentially a transition system extended with clock constraints. The notion of timed automata is introduced in Definition 2.4.1. The semantics of a timed automaton *TA* is formally defined by the transition system *TS*(*TA*) associated with it, which is introduced in Definition 2.4.2.

Definition 2.4.1 - Timed Automaton
A *timed automaton TA* is defined as the tuple (cf. [BK08], [AH99]):

$$TA = (Loc_{TA}, Loc_{TA}^{Init}, Act_{TA}, C_{TA}, Inv_{TA}, AP_{TA}, L_{TA}, E_{TA})$$

where:

- Loc_{TA} is a finite set of locations,
- $Loc_{TA}^{Init} \subset Loc_{TA}$ is a set of initial locations,
- Act_{TA} is a finite set of action labels,
- C_{TA} is a finite set of clocks,
- $Inv_{TA} : Loc_{TA} \rightarrow \mathcal{B}(C_{TA})$ is a function which maps each location s with an invariant from the set of clock constraints $\mathcal{B}(C_{TA})$,
- AP_{TA} is a finite set of atomic propositions,
- $L_{TA} : Loc_{TA} \rightarrow 2^{AP_{TA}}$ is a labeling function, and
- $E_{TA} \subset Loc_{TA} \times \mathcal{B}(C_{TA}) \times Act_{TA} \times 2^{C_{TA}} \times Loc_{TA}$ is a set of switches (s, ψ, a, D, s').

Each switch $(s, \psi, a, \lambda, s')$ is an edge from location $s \in Loc_{TA}$ to location $s' \in Loc_{TA}$ on action $a \in Act_{TA}$. The switch is enabled when the clock constraint $\psi \in \mathcal{B}(C_{TA})$ evaluates to the truth value *true*. The set of clocks $D \subseteq C_{TA}$ specifies which clocks are to be reset when the switch is enabled. ■

Definition 2.4.2 - Formal Operational Semantics for Timed Automata
Let *TA* be a timed automaton as in Definition 2.4.1. The *operational semantics of the timed automaton* TA is given by the transition system associated with it, which is defined as the tuple (cf. [BK08], [AH99]):

$$TS(TA) = (S_{TS}, S_{TS}^{Init}, Act_{TS}, AP_{TS}, L_{TS}, E_{TS})$$

where:

- S_{TS} is a finite set of states (s, ν), with $s \in Loc_{TA}$ and ν is clock interpretation over the set of clocks C_{TA},
- $S_{TS}^{Init} \subseteq S_{TS}$ is a set of initial states (s_0, ν_0), with $s_0 \in Loc_{TA}^{Init}$ and $\nu_0(x) = 0$, for all $x \in C_{TA}$,
- $Act_{TS} = Act_{TA} \cup \mathbb{R}_{\geq 0}$ is a finite set of action labels for the discrete transitions between locations along with real-valued time increments for the delay transitions,
- $AP_{TS} = AP_{TA} \cup \mathcal{B}(C_{TA})$ is a set of atomic propositions along with the clock constraints of the timed automaton,
- $L_{TS} : S_{TS} \to 2^{AP_{TS}}$ is a labeling function, where $L_{TS}((s, \nu))$ is the set $L_{TA} \cup \{\phi_{Clk} \in \mathcal{B}(C_{TA}) \mid \nu \models \phi_{Clk}\}$, and
- E_{TS} is the transition relation, defined through the following two rules:
  - *delay transitions:* for a state $(s, \nu) \in S_{TS}$ and a value $\delta \in \mathbb{R}_{\geq 0}$, the transition $(s, \nu) \xrightarrow{\delta} (s, \nu + \delta)$ takes place, if for all δ' with $0 \leq \delta' \leq \delta$, the clock interpretation $\nu + \delta'$ satisfies the invariant $Inv_{TA}(s)$, i.e., $\nu + \delta' \models Inv_{TA}(s)$,
  - *discrete transitions:* for a state $(s, \nu) \in S_{TS}$ and an action label $a \in Act_{TS}$, the transition $(s, \nu) \xrightarrow{a} (s', \nu[D := 0])$ takes place if there is a switch $(s, \psi, a, D, s') \in E_{TA}$ in the timed automaton, for which $\nu \models \psi$ and $\nu[D := 0] \models Inv_{TA}(s')$.

■

Definition 2.4.3 - Parallel Composition of Timed Automata
Let TA_1 and TA_2 be two timed automata as in Definition 2.4.1:

$$TA_1 = (Loc_{TA_1}, Loc_{TA_1}^{Init}, Act_{TA_1}, C_{TA_1}, Inv_{TA_1}, E_{TA_1}, AP_{TA_1}, L_{TA_1})$$
$$TA_2 = (Loc_{TA_2}, Loc_{TA_2}^{Init}, Act_{TA_2}, C_{TA_2}, Inv_{TA_2}, E_{TA_2}, AP_{TA_2}, L_{TA_2})$$

for which $C_{TA_1} \cap C_{TA_2} = \emptyset$ and $AP_{TA_1} \cap AP_{TA_2} = \emptyset$.

Then, the *parallel composition* of TA_1 and TA_2, also denoted as the product automaton, is defined as the tuple:

$$TA_1 || TA_2 = (Loc, Loc_{Init}, Act, C, Inv, E, AP, L)$$

where $Loc = Loc_{TA_1} \times Loc_{TA_2}$, $Loc_{Inv} = Loc_{TA_1}^{Init} \times Loc_{TA_2}^{Init}$, $Act = Act_{TA_1} \cup Act_{TA_2}$, $C = C_{TA_1} \cup C_{TA_2}$, $Inv((s_1, s_2)) = Inv_{TA_1}(s_1) \wedge Inv_{TA_2}(s_2)$, $AP = AP_{TA_1} \cup AP_{TA_2}$, and $L((s_1, s_2)) = L_{TA_1}(s_1) \cup L_{TA_2}(s_2)$. The switches in the product automaton are defined by the following rules:

- for action $a \in Act_{TA_1} \cap Act_{TA_2}$, for every switch $(s_1, \psi_1, a, D_1, s_1')$ in E_{TA_1} and $(s_2, \psi_2, a, D_2, s_2')$ in E_{TA_2}, there exists the switch $((s_1, s_2), \psi_1 \wedge \psi_2, a, D_1 \cup D_2, (s_1', s_2'))$ in E,
- for action $a \in Act_{TA_1} \setminus Act_{TA_2}$, for every switch $(s_1, \psi_1, a, D_1, s_1')$ in E_{TA_1} and $s_2 \in Loc_{TA_2}$, there exists the switch $((s_1, s_2), \psi_1, a, D_1, (s_1', s_2))$ in E,
- for action $a \in Act_{TA_2} \setminus Act_{TA_1}$, for every switch $(s_2, \psi_2, a, D_2, s_2')$ in E_{TA_2} and $s_1 \in Loc_{TA_1}$, there exists the switch $((s_1, s_2), \psi_2, a, D_2, (s_1, s_2'))$ in E.

■

Notice that in a product of two timed automata, a component timed automaton synchronizes with the other timed automaton over the switches that have identical action labels and executes independently of the other component automaton those switches that have different labels.

Uppaal is a verification tool which implements the formalism of timed automata introduced by Alur in [AD92] as Uppaal automata. It provides a graphical language for modeling the Uppaald a textual language similar to the C programming language which allows modeling the behavior of the Uppaal automata with the help of user defined variables and functions. In Uppaal, the parallel composition between timed automata results in a network of timed automata (cf. [BY04]), which constitutes the Uppaal model used to describe the system model and the environment model. The parallel composition between two Uppaal automata is realized through channels, which are used for synchronous communication between the two automata. Uppaal implements both synchronous and asynchronous communication between timed automata. The synchronous communication is realized via an alphabet of input and output actions, which are implemented as channels, while the asynchronous communication occurs through shared variables (cf. [BY04]). Consider two Uppaal automata TA_1 and TA_2 that synchronize over a channel m. Timed automaton TA_1 contains a transition labeled with the output action $m!$ and timed automaton TA_2 has a transition labeled with the input action $m?$. The alphabet of actions consists of the input action $m?$ and output action $m!$. If the transition labeled with the action $m!$ is enabled in TA_1 and the corresponding transition labeled with action $m?$ is enabled in TA_2, then TA_1 and TA_2 synchronize with each other and execute together their respective transitions, e.g., TA_1 outputs a computation result and TA_2 reads it and uses it further in its own computations.

2.4.2. Short Overview of Modeling Formalisms for Probabilistic Systems

There is a wide range of formalisms, that can be used to model the behavior of probabilistic safety-critical systems. Prism and Storm are two verification tools which implement some of these formalisms (cf. [KNP02], [DJKV17]). An overview of the formalisms used to describe probabilistic systems in the model checker Storm is given in Table 2.2.

The models in Table 2.2 are categorized according to two dimensions: model of time and determinism or lack thereof. Discrete-time Markov chains (DTMCs) are considered the simplest of the probabilistic models. A DTMC is a transition system in which each transition has a probability associated with it and for each state the sum of probabilities for the outgoing transitions equals to 1 (cf. [Kat16]). In contrast to DTMCs which have a discrete model of time, in continuous-time Markov chains (CTMCs) time is continuous. Each state in a CTMC has a negative exponential distribution associated with it, that defines the residence time in the respective state. Similar to a DTMC, there is a probability distribution over the successor states for each state of a CTMC (cf. [Kat16]).

A Markov decision process (MDP) extends a DTMC with nondeterminism. Each state may have several outgoing transitions, each labeled with an action. In order to switch to another state, a transition labeled with a unique action is chosen in a nondeterministic

Table 2.2.: Overview of some Formalisms used for Modeling of Probabilistic Systems (cf. [Kat16]).

	Discrete Time	**Continuous Time**
Deterministic	Discrete-time Markov Chain (DTMC)	Continuous-time Markov Chain (CTMC)
Nondeterministic	Markov Decision Process (MDP)	Continuous-time Markov Decision Process (CTMDP)
Compositional	Probabilistic Automata (PA)	Markov Automata (MA)

manner, after which the successor state is elected according to a probability distribution (cf. [Kat16]). A scheduler or a policy prescribes which probability distribution is selected when an MDP is in a given state. In this way, a scheduler transforms an MDP in a DTMC (cf. Section 2.4.3).

Continuous time Markov decision processes (CTMDPs) extend MDPs with the notion of continuous time as it is modeled in CTMCs, i.e., the time spent in a state of a CTMDP is defined through a negative exponential distribution. This distribution depends on the probability distribution that is selected and used to determine the next state (cf. [Kat16]).

In addition to the formalisms presented above, there are also compositional variants of them. The compositional formalisms are extensions of the presented Markov models, which can be used to describe complex probabilistic systems using parallel composition with process-algebraic operators like in calculus of communicating systems (CCS) or in communicating sequential processes (CSP) (cf. [Kat16]). Thus, probabilistic automata (PA) introduced by Segala in [Seg95] are an extension of MDPs, in which the probability distributions in state are labeled with actions. Parallel composition of PA is realized through synchronization between the two component PA over a set of common actions. Each PA executes the actions outside of this set independently and in an interleaved manner (cf. [Kat16]). Markov automata (MA) are an extension of CTMDPs, and from a technical point of view MA are similar to PA with the additional feature that transitions can be labeled with actions or with positive real numbers, which represent rates of exponential distributions. Transitions labeled with rates in an MA can be understood as delay transitions (cf. [Kat16]). Semantically, an MA combines the behavior of CTMC with that of a PA. In states with the outgoing transitions labeled by actions, an MA behaves as a PA, while in states with outgoing transitions labeled by rates the MA works as CTMC (cf. [Kat16]).

The Prism model checker supports all of the formalisms in Table 2.2 with the exception of MA and CTMDPs. In addition to the formalisms it has in common with the Storm model checker, the Prism model checker also supports probabilistic timed automata (PTA), as well as variants of MDP and PTA in which the scheduler that resolves the nondeterminism in the model has access only to observations of the model state, instead of accessing its actual state.

2.4.3. Markov Decision Processes

Markov decision processes, introduced first in [Bel57], [How60] and [Put94], are used to represent systems which exhibit probabilistic as well as nondeterministic behavior (cf. [FKNP11]), and in which the evolution of the system under analysis is encoded by discrete probabilities (cf. [BdAFK18]). In MDPs, transitions are labeled with actions, which can be chosen in a nondeterministic fashion. The successor states for the chosen action are specified through discrete probability distributions. In this respect, MDPs can be considered as a probabilistic variant of labeled transition systems (cf. [BdAFK18]). Throughout this thesis, MDPs are considered to be equivalent to PA (cf. [BdAFK18]), due to Segala's work (cf. [Seg95]). Before the introducing the notion of Markov decision process it is important to define the notion of probability distribution and the product of two probability distributions. This is done in Definition 2.4.4 and Definition 2.4.5. The notion of Markov decision process is introduced in Definition 2.4.6.

Definition 2.4.4 - Probability Distribution
Let X be a countable set. A *probability distribution* over the set X is a function $D : X \to [0,1]$ for which $\sum_{x \in X} D(x) = 1$ holds (cf. [BdAFK18]). ■

Notation 2.4.1. $\mathit{Supp}(D) \overset{\text{def}}{=} \{x \in X : D(x) \neq 0\}$ is denoted the support set of the distribution D and contains all elements of X which have associated a nonzero probability with them. If the support set of the probability distribution D contains only a single element, then D is denoted as a *Dirac* distribution. $\mathit{Distr}(X)$ denotes the set of all distributions over set X (cf. [BdAFK18]). ■

Definition 2.4.5 - Product of Probability Distributions
Let X_1 and X_2 be two countable sets. Let $D_1 \in \mathit{Distr}(X_1)$ and $D_2 \in \mathit{Distr}(X_2)$ be two probability distributions over these sets. Then, the *product of the two probability distributions* is denoted as $D_1 \times D_2 \in \mathit{Distr}(X_1 \times X_2)$ and is defined by $D_1 \times D_2((x_1, x_2)) = D_1(x_1) \cdot D_2(x_2)$ (cf. [FKNP11]). ■

Definition 2.4.6 - Markov Decision Process
A *Markov decision process (MDP)* is defined as the tuple:

$$\mathcal{M} = (S, s_{init}, Act, \mathbf{P}, AP, L)$$

where:

- S is the countable non-empty set of states of $\mathcal{M}$,
- s_{init} is the initial state of $\mathcal{M}$,
- Act is the finite non-empty set of actions of $\mathcal{M}$,
- $\mathbf{P} : S \times Act \times S \to [0,1] \cap \mathbb{Q}$ is the transition probability function of $\mathcal{M}$, such that

$$\sum_{s' \in S} \mathbf{P}(s, a, s') \in \{0, 1\},$$

  for all system states $s \in S$ and for all system actions $a \in Act$,

- AP is a finite set of atomic propositions in $\mathcal{M}$,
- $L : S \to 2^{AP}$ is a labeling function of $\mathcal{M}$ which labels a state $s \in S$ with the atomic propositions $a \in Act$ that are supposed to hold in s. ■

Notation 2.4.2. $Act(s)$ denotes the set of actions which are enabled in state the s of $\mathcal{M}$. An action $a \in Act$ is enabled in state s of $\mathcal{M}$ if and only if $\sum_{s' \in S} \mathbf{P}(s, a, s') = 1$, i.e., at least one outgoing transition of state s labeled with the action a has a nonzero probability associated with it. No state s in $\mathcal{M}$ is allowed to be a terminal state, i.e., $Act(s) \neq \emptyset$ (cf. [BK08], [BdAFK18]). This is in order to prevent deadlocks, since $Act(s) \neq \emptyset$ means that there is always an action enabled in the state s of $\mathcal{M}$ which can be taken in order for $\mathcal{M}$ to progress to the next state (cf. [FKNP11]) ■

The definition for MDP used in this thesis is adapted from the one given by Baier et al. in [BdAFK18]. In comparison with the definition given in [BdAFK18], the one used in this thesis does not include the cost function associated with executing a path or a set of paths in a Markov decision process. An example in which the definition of a cost function is necessary is an autonomous robot which aims to reach a given destination as soon as possible. However, in this thesis, MDPs are used to model automated safety-critical systems that operate in uncertain environments. For such systems, the purpose is to express safety properties that ask for example the probability with which the system under analysis remains in a safe state, given the probability of occurrence for a hazardous event. In order to be able to reason about MDPs, their operational semantics must be understood. Definition 2.4.7 introduces the operational semantics of MDPs at an intuitive level.

Definition 2.4.7 - Intuitive Operational Semantics for Markov Decision Processes
Let $\mathcal{M}$ be an MDP as defined in Definition 2.4.6. Its *operational semantics* can be described intuitively as follows. The MDP $\mathcal{M}$ starts its computation in its initial state s_{init}. After n computation steps, the current state of $\mathcal{M}$ is s_n and $\mathcal{M}$ chooses the action $a_{n+1} \in Act(s_n)$ in a nondeterministic manner. The effect of taking action a_{n+1} in s_n is controlled by the probability distribution $\mathbf{P}(s_n, a_{n+1}, \cdot)$. The next state s_{n+1} is an element of the support set of $\mathbf{P}(s_n, a_{n+1}, \cdot)$ and it is chosen based on its probability. The execution of $\mathcal{M}$ results in a possibly infinite sequence of states and actions of the form $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots$, with $s_{init} = s_0$ (cf. [BdAFK18]). ■

Notation 2.4.3. Let $\mathcal{M}$ be an MDP. A path of state s_0 in $\mathcal{M}$ is an infinite alternating sequence of states and actions of the form $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots$, in which $s_i \in S$, $a_{i+1} \in Act(s_i)$, and $\mathbf{P}(s_i, a_{i+1}, s_{i+1}) > 0$, with $i \geq 0$ (cf. [BdAFK18]).

A finite path of state s_0 in $\mathcal{M}$ has the form $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$. For the finite path π, $last(\pi)$ denotes the last state of the path π and $|\pi|$ is called the length of the path π (cf. [FKNP11]).

For an infinite or a finite path π in $\mathcal{M}$, $first(\pi)$ denotes the first state of the path π and $\pi[i]$ denotes the $(i+1)$-th state in the path π, i.e., $\pi[i] = s_i$ (cf. [BdAFK18]).

There are several sets of paths which are defined with respect to a state s in $\mathcal{M}$: $Paths_{\mathcal{M}}(s)$ denotes the set of all paths in $\mathcal{M}$ starting in the state s, while $FinPaths_{\mathcal{M}}(s)$

is the set of all finite paths in $\mathcal{M}$ starting in the state s. With respect to the whole MDP $\mathcal{M}$, following sets of paths are defined: $\mathit{Paths}_{\mathcal{M}}$ denotes the set of all paths in $\mathcal{M}$, while $\mathit{FinPaths}_{\mathcal{M}}$ represents the set of all finite paths in $\mathcal{M}$ (cf. [BdAFK18]). ■

In order to reason about probabilities in MDPs, it is important to have a mechanism which resolves the nondeterministic choices, which exist between the transitions of an MDP (cf. [BdAFK18]). This is done through a decision making approach which chooses in each state of the MDP which action is to be performed, based on the history of the MDP's execution up to the respective state (cf. [FKNP11]). The decision making approach is formally modeled using the notion of scheduler. In literature, schedulers are referred to as adversaries, policies or strategies (cf. [FKNP11], [BdAFK18]) The notion of scheduler is introduced in Definition 2.4.8.

Definition 2.4.8 - Scheduler
Let $\mathcal{M}$ be an MDP as defined in Definition 2.4.6. Then, a *scheduler* is a function $\mathcal{U} : \mathit{FinPaths}_{\mathcal{M}} \rightarrow \mathit{Distr}(\mathit{Act})$ such that, for all finite paths π in $\mathcal{M}$, $\mathit{Supp}(\mathcal{U}(\pi)) \subseteq \mathit{Act}(\mathit{last}(\pi))$ (cf. [BdAFK18]). ■

Notice that what Definition 2.4.8 expresses is that all actions with nonzero probability in the scheduler $\mathcal{U}$ are enabled in the last state of the finite path π, and one of them can be chosen, based on its probability, to be executed next. Intuitively, a scheduler $\mathcal{U}$ takes a finite path π, i.e., a history of computation in the MDP $\mathcal{M}$, as input and, based on a probability distribution, chooses the next action to be executed (cf. [BdAFK18]). By assigning probabilities to the nondeterministic choices available in the last state of π, scheduler $\mathcal{U}$ transforms the MDP $\mathcal{M}$ in a DTMC (cf. [BK08], [ÁBD+14]).

Notation 2.4.4. A finite or infinite path $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots$ in an MDP $\mathcal{M}$ is denoted as a path of a scheduler $\mathcal{U}$ if, for any path fragment $\pi_i = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots \xrightarrow{a_i} s_i$, there exists an enabled action a_{i+1} in π which can be chosen by the scheduler $\mathcal{U}$, i.e., $\mathcal{U}(\pi_i)(a_{i+1}) > 0$ (cf. [BdAFK18]). $\mathit{Sched}_{\mathcal{M}}$ denotes the set of all schedulers in an MDP $\mathcal{M}$. ■

The notion of scheduler as defined in Definition 2.4.8 describes a history-dependent randomized scheduler. Schedulers are categorized depending on which information they use in order to make a decision and whether randomization is used or not. Besides history-dependent randomized schedulers, there are deterministic schedulers and memoryless schedulers.

Notation 2.4.5. A deterministic scheduler is a function $\mathcal{U} : \mathit{FinPaths}_{\mathcal{M}} \rightarrow \mathit{Act}$, where for all finite paths $\pi \in \mathit{FinPaths}_{\mathcal{M}}$, $\mathcal{U}(\pi)$ is a Dirac probability distribution. This means that a deterministic scheduler selects always some action a with probability 1, while all the other actions have probability 0 (cf. [BdAFK18]).

A memoryless randomized scheduler is a function $\mathcal{U} : S \rightarrow \mathit{Distr}(\mathit{Act})$, for which $\mathcal{U}(s_i)(a_{i+1}) > 0$, where $a_{i+1} \in \mathit{Act}(s_i)$, $s_i = \mathit{last}(\pi_i)$, and $\pi_i \in \mathit{FinPaths}_{\mathcal{M}}$. This means that a memoryless scheduler makes a decision based only on the current state, rather than considering the states encountered previously as history-dependent schedulers do. A memoryless deterministic scheduler, also denoted as simple scheduler, is represented by a function $\mathcal{U} : S \rightarrow \mathit{Act}$ (cf. [BdAFK18]). ■

Using MDPs to model an automated safety-critical system and its environment means that the parallel composition for MDPs must also be defined. This is done in Definition 2.4.9, which is inspired by the definition of parallel composition for PA given by Segala in [Seg95].

Definition 2.4.9 - Parallel Composition of Markov Decision Processes
Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two MDPs as in Definition 2.4.6:

$$\mathcal{M}_1 = (S_1, Act_1, \mathbf{P}_1, s^1_{init}, AP_1, L_1)$$
$$\mathcal{M}_2 = (S_2, Act_2, \mathbf{P}_1, s^2_{init}, AP_2, L_2)$$

Then, the parallel composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ results in an MDP, which is represented by the tuple:

$$\mathcal{M}_1 || \mathcal{M}_2 = (S, Act, \mathbf{P}, s_{init}, AP, L)$$

where:

- $S = S_1 \times S_2$ is the countable non-empty set of states of the composite MDP,
- $Act_1 = Act_1 \cup Act_2$ is the finite non-empty set of actions in the composite MDP,
- $\mathbf{P} : S \times Act \times S \to [0,1] \cap \mathbb{Q}$ is the transition probability function such that
$$\sum_{s' \in S} \mathbf{P}(s, a, s') \in \{0, 1\}$$
for all states $s \in S$ and for all actions $a \in Act$ in the composite MDP,
- $s_{init} = (s^1_{init}, s^2_{init})$ is the initial state of the composite MDP,
- $AP = AP_1 \cup AP_2$ is a finite set of atomic propositions,
- $L : S \to 2^{AP}$ is a labeling function which labels a state $s = (s_1, s_2)$ of the composite MDP with the atomic propositions which are supposed to hold in this state.

The parallel composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ is done using the CSP-based operator $||$. The transitions of the composite MDP and their respective probabilities are defined according to the following rules:

- for action $a \in Act_1 \setminus Act_2$, for every transition (s_1, a, s'_1) in $\mathcal{M}_1$, transition $((s_1, s_2), a, (s'_1, s_2))$ is added to $\mathcal{M}$, with the probability
$$\mathbf{P}((s_1, s_2), a, (s'_1, s_2)) = \mathbf{P}_1(s_1, a, s'_1)$$
- for action $a \in Act_1 \setminus Act_2$, for every transition (s_2, a, s'_2) in $\mathcal{M}_2$, transition $((s_1, s_2), a, (s_1, s'_2))$ is added to $\mathcal{M}$, with the probability
$$\mathbf{P}((s_1, s_2), a, (s_1, s'_2)) = \mathbf{P}_2(s_2, a, s'_2)$$
- for action $a \in Act_1 \cap Act_2$, for every transition (s_1, a, s'_1) in $\mathcal{M}_1$ and (s_2, a, s'_2) in $\mathcal{M}_2$, transition $((s_1, s_2), a, (s'_1, s'_2))$ is added to $\mathcal{M}$, with the probability
$$\mathbf{P}((s_1, s_2), a, (s'_1, s'_2)) = \mathbf{P}_1(s_1, a, s'_1) * \mathbf{P}_2(s_2, a, s'_2)$$

■

Prism and Storm are two verification tools which implement a wide range of formalisms, that can be used to model the behavior of probabilistic systems (cf. Section 2.4.2). The two verification tools share a common textual modeling language, called Prism, which is a state-based language inspired by the formalism of reactive modules introduced by Alur and Henzinger in [AH99] (cf. [HKNP06]). A Prism model consists usually of several modules, whose behavior is described by a set of commands. A command in the Prism modeling language has the general form shown in Equation (2.1):

$$[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m \tag{2.1}$$

where g is the guard of the Prism command and $u_1, \ldots u_m$ are several updates that can be carried out with different probabilities $\lambda_1 \ldots \lambda_m$. Optionally, a Prism command is labeled with an action a.

Prism supports several CSP-based operators which allow the modeling of several types of parallel composition between two or more MDPs through (1) synchronization over a common alphabet of actions with interleaving on all the other actions that are not included in the alphabet, (2) full interleaving with no synchronization, (3) renaming of actions in one or several modules, and (4) hiding of actions in one or several modules.

In this thesis, the parallel composition of two MDPs is realized through synchronization over the common actions between the two MDPs and interleaving for all the other actions, that do not belong to the common alphabet. Given two MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$, their parallel composition is done according to the following rules[2]:

- for each action $a \in Act_1 \setminus Act_2$ and command $[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m$ of $\mathcal{M}_1$ add command $[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m$ to $\mathcal{M}_1 \mid\mid \mathcal{M}_2$,
- for each action $a \in Act_2 \setminus Act_1$ and command $[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m$ of $\mathcal{M}_2$ add command $[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m$ to $\mathcal{M}_1 \mid\mid \mathcal{M}_2$,
- for each action $a \in Act_1 \cap Act_2$, command $[a]\ g \rightarrow \lambda_1 : u_1 + \cdots + \lambda_m : u_m$ of $\mathcal{M}_1$ and command $[a]\ g' \rightarrow \gamma_1 : v_1 + \cdots + \gamma_r : v_r$ of $\mathcal{M}_2$ add the command

$$\begin{aligned}[a]\ g \ \&\ g' \rightarrow & \lambda_1 * \gamma_1 : u_1 \ \&\ v_1 + \ldots \lambda_n * \gamma_1 : u_n \ \&\ v_1 \\ & + \lambda_1 * \gamma_2 : u_1 \ \&\ v_2 + \ldots \lambda_n * \gamma_2 : u_n \ \&\ v_2 \\ & \vdots \\ & + \lambda_1 * \gamma_r : u_1 \ \&\ v_r + \ldots \lambda_n * \gamma_r : u_n \ \&\ v_r \end{aligned}$$

  to $\mathcal{M}_1 \mid\mid \mathcal{M}_2$.

2.5. Verification and Validation of System Properties in Automated Safety-critical Systems

The verification and validation process accompanies the system development process of an automated safety-critical system starting with the concept phase, in which a safety

[2] https://www.prismmodelchecker.org/doc/semantics.pdf

concept is defined along the functional concept of the system under development, and ending with the acceptance tests and the safety assessment of the system carried out before releasing it for production (cf. Section 2.2). Verification and validation are two types of activities that are carried out with different goals in mind during the development of automated safety-critical systems. The notions of verification and validation as they are understood throughout this thesis are introduced in Definition 2.5.1 and Definition 2.5.2.

Definition 2.5.1 - Verification
Verification is the process of evaluating a system or a system component in order to determine whether the artifacts obtained as a result of a given development phase satisfy the conditions imposed at the start of that phase (cf. [ISO10]). The verification process is aimed at one single development phase at a time and its purpose is to provide objective evidence that the outcome of the respective development phase is achieved correctly and completely with respect to its specification, i.e. the requirements and conditions imposed through specific input documents at the start of the phase (cf. [SLS14c]). ■

Definition 2.5.2 - Validation
Validation is the process of ensuring that a developed system is able to accomplish its intended use, goals and objectives (cf. [ISO10]). The validation process can take place at different levels of abstraction (cf. [SLS14c]) and evaluates the system or a system component in order to check whether the system or the system component satisfies the user needs and those of other identified stakeholders (cf. [ISO10]). ■

Notice that both verification and validation can take place at any level during the system development process. The difference between the two is that verification is concerned with checking the correctness of a developed system with respect to a given specification, focusing on answering the question whether the system is correctly built or not, while in comparison, validation focuses on checking whether the developed system fulfills its intended purpose and meets the user needs, thus answering the question whether the right system was built (cf. [SLS14c]).

There are various methods by which verification and validation of automated safety-critical systems can be carried out. This section gives an overview of three of these methods, which are used in this thesis - testing (Section 2.5.1), design-time verification (Section 2.5.2), and runtime verification (Section 2.5.3) - and introduces some core notions associated with them.

2.5.1. Testing

Testing is the principal approach of verification and validation recommended by ISO 26262 and ISO/PAS 21448 (cf. Section 2.2). Testing accompanies the development phases of a safety-critical system and specific test methods are integrated in the development process at system level as well as at hardware and software level in order to verify that the system as a whole as well as the individual hardware and software components comply with their respective functional and safety requirements (cf. Section 2.2). The notion of testing as it is understood and used throughout this thesis is introduced in Definition 2.5.3.

Definition 2.5.3 - Testing
Testing is defined as an activity in which a system or a system component is executed under predefined conditions, the results are observed or recorded, and specific aspects of the system or of the system component are evaluated based on the recorded results (cf. [ISO10]). An artefact under test, e.g., a system or a system component, is denoted as a *test object.* ■

In the software development process presented in ISO 26262, there are three large test phases: software unit test, software integration test and software safety requirements verification (cf. Section 2.2.1). In order to test a particular test object, test engineers define test cases as finite sequences of test inputs and test outputs. During testing, the test object is stimulated with the defined test inputs and the computations' result is compared to the expected value defined by the test engineers. While software unit tests check atomic software components individually with respect to their design specification, software integration tests accompany the software integration process and check the test object and its interfaces against the software architectural design (cf. Section 2.2.1). The software safety requirements verification can be seen as a system test for the integrated software system, which is checked in its target environment with respect to its functional and safety requirements. As the process of software and system integration progresses, the environment of the test object consists of other software components as well as software and hardware components and subsystems. Software unit tests as well as software integration tests can be carried out in various environments, depending on the level at which the tests are performed. Such tests are simulation-based tests, called *X-in-the-loop* tests, where X stands in for a model, a piece of software, a target processor or a target hardware (cf. Section 2.2.1). The environment of a test object which is part of a larger automated safety-critical system is a technical one, consisting of software and/or hardware components. At higher levels of development and integration, e.g., at system level, the test object is not a system component or a subsystem anymore but the whole system itself. In this case, the environment of the test object is in fact the physical environment in which the system is deployed, which is part of the physical world. Definitions 2.5.4 and 2.5.5 introduce the notions of technical environment and physical environment as they are understood and used henceforth in this thesis.

Definition 2.5.4 - Technical Environment
A *technical environment* is defined in relation to a test object which is part of a larger automated safety-critical system. The technical environment consists of software and/or hardware components with which the test object communicates through its software and hardware interfaces. ■

Definition 2.5.5 - Physical Environment
A *physical environment* is defined in relation to an automated safety-critical system, of which the test object is a part of. The physical environment consists of dynamic elements,

scenery elements, self-representation of the system itself and of other actors present in the environment as well as the relationships between all these entities. ■

Notice that the concept of physical environment is related to the notion of scene introduced in Section 2.2.2, since a scene is considered to be a snapshot of the physical environment in which the system under test is deployed (cf. Definition 2.2.6). To give an example of the elements that may be contained in the physical environment of an automated safety-critical system, consider the case of an autonomous vehicle. The dynamic elements in the physical environment may be represented by moving obstacles, e.g., other vehicles. The scenery may contain information on the infrastructure elements present in the physical environment, the geometry of the environment, and environment conditions, e.g., traffic lights, number of lanes, and weather conditions.

In order to test a test object, test engineers create a series of test cases, which are executed on the test object. The notion of test case as it is understood in this thesis is given in Definition 2.5.6. Related to the execution of test cases on a test object are the notions of system trace and system execution, introduced in Definitions 2.5.7 and 2.5.8.

Definition 2.5.6 - Test Case
A *test case* is a set of inputs, a precondition, an expected result, and a postcondition, defined for a test object with a specific objective in mind, e.g., exercise a particular path in an algorithm or verify compliance with a specific requirement (cf. [SLS14c], [ISO10]). The test inputs are also denoted as stimuli, while the outputs produced by the test object are denoted as responses. ■

Definition 2.5.7 - System Trace
Let S be a test object. A *system trace* of S is a possibly infinite sequence of system states of S or a sequence of actions performed by S (cf. [LS09]). ■

Definition 2.5.8 - System Execution
Let S be a test object. A *system execution* of S is a finite trace of S (cf. [LS09]). ■

Notice that state of a test object can be regarded as a snapshot of the test object at a certain step during its execution. A state is in fact a interpretation of the state variables over their respective value domains. When running, the test object performs certain actions. Each action starts with a specific interpretation of the state variables and produces a new interpretation of them, i.e. the state variables are assigned specific values from their respective value domains as a result of the action carried out by the test object.

Given a test object S and a test case tc, the precondition describe the state of S before the execution of S with the test case tc, while the postcondition expresses the expected state of S afterwards. The postcondition compares the expected result with the output produced by the test object S. The postcondition is sometimes referred to as a test oracle. The notion of test oracle as it is understood in this thesis is defined in Definition 2.5.9.

Definition 2.5.9 - Test Oracle
Let S be a test object and tc a test case specified for S. A *test oracle* is defined as a boolean function from a system execution of the test object S to the boolean set $\{\text{true}, \text{false}\}$. The test oracle determines whether the execution of S with the test case tc is correct, by comparing the expected output defined by the test case tc with the output produced by S during its execution. ■

Notice that Definition 2.5.9 aligns with the definitions introduced by Richardson et al. [RAO92] and Barr et al. [BHM$^+$15]. Barr et al. define a test oracle as a partial boolean function over a set of stimuli and responses of the test object (cf. [BHM$^+$15]). In turn, Richardson et al. consider that a test oracle consists of two parts: oracle information and oracle procedure. The oracle information defines what constitutes correct behavior of the test object S, while the oracle procedure verifies the results of the test case execution with regard to the respective oracle information (cf. [RAO92]).

There are various methods for specifying a test oracle. The oracle information can be extracted from the requirements specification of the test object S, from a reference implementation of S that is available as an executable code or model, or it can be manually defined by experts (cf. [SWH11]). If a property specification ϕ holds for the system S when executing the test case tc, then it is said that the system S has successfully passed the test case tc (cf. [SWH11]). The oracle procedure can be realized as mechanism which monitors inputs with which the test object S is stimulated and its outputs and compares the output computed by S with the expected output specified in the oracle information. Another form of test oracles are assertions defined by experts in the test object S, which do not verify only the final result but also the intermediate results of a system execution of the test object S (cf. [RAO92]).

Test cases may be defined manually by experts, but they can also be automatically generated using design-time formal verification methods such as model checking. Various methods are recommended by ISO 26262 for the manual definition of test cases for the software unit tests and the software integration tests, e.g., definition and analysis of equivalence classes, boundary value analysis as well as analysis of the functional and safety requirements of the respective test object (cf. Section 2.2.1).

Used as a method for the verification and validation of automated safety-critical systems, testing can show the presence of faults or defects in a test object but never their absence (cf. [Dij72]). This is because testing is incomplete and can never cover the entire set of possibly reachable states of a test object. Figure 2.9 gives a visual intuition of testing.

In an effort to increase the relevance of the defined test cases and also to ensure their traceability to the requirements of the test object, test engineers can make use of automated methods such as model checking in order to automatically generate requirements-based test cases. One way to generate test cases using model checking is to build trap properties (cf. [AHDR18]), which are essentially negations of the original system requirements (cf. [AVR19a]). This approach has been shown to work in the aeronautics domain (cf. [WRHM06], [SWRH10]), and has been later transferred to the application domain of automotive control systems (cf. [AHDR18]). The premise for using model

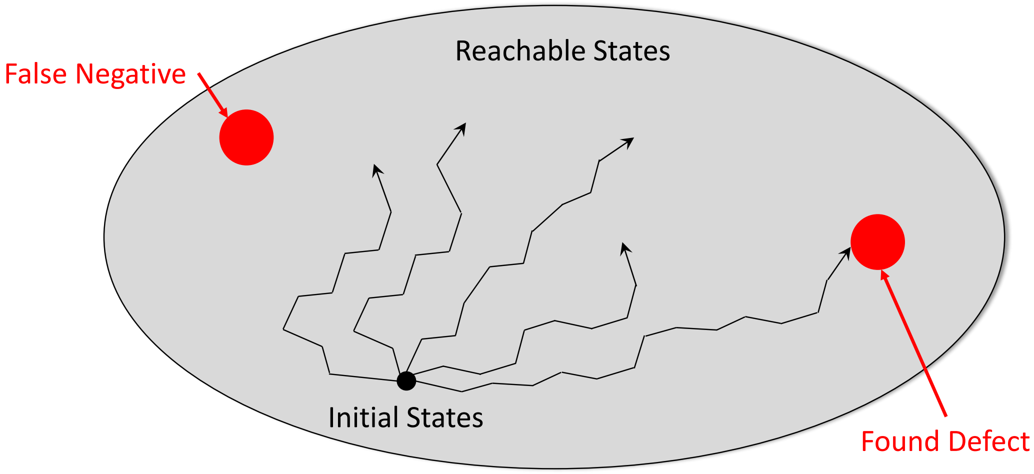


Figure 2.9.: Visual Intuition of Testing.

checking is the existence of a formal model for the test object and a formal specification for its requirements (cf. Section 2.5.2) and for the respective trap properties. The formal model of the test object is built correctly with respect to its original requirements, which is shown through the design-time verification methods applied during its design. The automated generation of test cases via model checking relies on the basic working principle of this method, i.e. when verifying a formal model against a formal property specification, the model checker searches the reachable state space of the model for a counterexample which disproves the property specification (cf. Section 2.5.2). If a counterexample is found when verifying the formal model of the test object against a trap property, then by the law of double negation in propositional logic, the counterexample which disproves the trap property is one that satisfies the original requirement of the test object (cf. [AHDR18], [AVR19a]).

Notice that the form of testing introduced in this section is also denoted *oracle-based testing*.

2.5.2. Design-time Verification

Along with testing, ISO 26262 recommends that formal verification methods be applied at design-time to provide proof of correctness of software components with higher ASIL with respect to their respective software safety requirements (cf. Section 2.2.1). For safety-critical systems in the aeronautics domain, the standard RTCA DO-178C recognizes that verification is not reduced to testing and instead regards the activities in the verification process as a combination of reviews, tests, and analyses of the system under development (cf. [RTC11]).

Design-time verification is a complementary activity to testing that can be carried out at architectural level but also at implementation level during the system development process. Methods applied in design-time verification are used to carry out analyses of the system under development with respect to certain properties, which the system designers would like the system to fulfill. The notion of design-time verification as it is understood and used throughout this thesis is given in Definition 2.5.10.

Definition 2.5.10 - Design-time Verification
Design-time verification is an activity in which formal verification methods are used during design-time of a system under development in order to establish whether the system satisfies a given property. ■

Formal verification is the process of applying a manual or automated technique in order to ensure that a system satisfies a given property or behaves in accordance with some higher level description of it (cf. [Pel01]). Formal verification methods usually rely on mathematical theories, e.g., logic and automata (cf. [Pel01]), which allow the development of a formal model for the system under analysis as well as a formal specification of the system property to be verified. A formal model of a system is in fact a mathematical abstraction of that system, that simplifies the system description and preserves only the system aspects which the system designers want to analyze.

When applying design-time verification to an automated safety-critical system, system designers want to make sure that the system under development works properly in its operational design domain according to its functional and safety requirements. For this purpose, system designers do not create only a formal model of the system but also a formal model of the environment. These models are denoted as technical system model and environment model, two concepts which are formally introduced in Definitions 2.5.11 and 2.5.12.

Definition 2.5.11 - Technical System Model
A *technical system model* is defined as a formal abstract representation of a system under analysis. The technical system model reflects in an abstract manner one or more aspects of the system, which are considered relevant by system designers in process of design-time verification. A technical system model is connected with an environment model through a system-environment interface. ■

Definition 2.5.12 - Environment Model
An *environment model* is defined as a formal abstract representation of the environment of a system under analysis. Depending on the hierarchy level at which the system under analysis is situated, the environment model reflects in an abstract manner one or more aspects of the technical environment and/or of the physical environment, which are considered relevant by system designers in the process of design-time verification. An environment model is connected with a technical system model through the system-environment interface. ■

The technical system model and the environment model are combined together in an overall system model. During design-time verification, reasoning about the system with respect to a given property specification is done on the basis of the overall system model. This allows system designers to focus their analysis only on certain aspects of the system and its environment and thus manage the complexity of the system as well as that of its environment more efficiently.

Depending on the method used for the design-time verification, the analysis can be carried out manually, semi-automated or fully automated with the help of specialized tools.

Deductive Software Verification

Deductive software verification is one of the first formal verification techniques studied, which has been developed in various proof systems, that were applied for the verification of a program's correctness, e.g., Hoare calculus [Hoa69] (cf. [Pel01]). Deductive verification relies on the step-wise refinement of the program under analysis, starting from its formal specification and ending with the actual code of the program, so that at each refinement step the correctness of the step is preserved (cf. [Pel01]). Even though some parts of deductive verification can be automated, it remains primarily a manual technique which requires a lot of expertise and is applicable mainly on small examples (cf. [Pel01]).

Theorem Proving

Theorem proving is a semi-automated technique used to check a program's correctness by proving it as one would prove a mathematical theorem (cf. [Mau19]), starting with well-known axioms as premises and applying established proof rules in an iterative manner in order to obtain the desired proof goals. Theorem provers are tools used for obtaining and checking proofs, based on an underlying proof system, which consists of axioms and proof rules (cf. [Pel01]). Nevertheless, rather than applying one proof rule or one axiom at a time, the process of theorem proving can be accelerated by using proof-automation procedures or tactics, which combine the application of several axioms and proof rules (cf. [Pel01]).

Theorem proving tools are built to enforce rigor in the steps by which a proof is obtained. However, there are two approaches by which this is done: a so-called purist approach and a more engineering-oriented approach (cf. [Pel01]). Under the first approach the proofs are based on a small number of well-known axioms and proof rules. The user cannot add axioms about objects from new domains, but instead needs to prove theorems about them. In order to give support to the user, different mathematical theories that have already been proven are organized in libraries and are made available in theorem provers for reuse in future proofs (cf. [Pel01]), e.g., the theorem prover Isabelle/Hol which uses higher-order logic for the specification and verification of systems (cf. [Nip02]). Theorem provers using higher-order logic aim for expressiveness, i.e. the data manipulation is handled precisely, and for full functional correctness (cf. [CHV18]).

In the more engineering-oriented approach, the user is allowed to provide the axioms for the application domain in which the system under analysis is situated, but he is also responsible to ensure that the axioms reflect the properties of the intended domain (cf. [Pel01]). With this approach, users may add more axioms in order to make the proof more easier to obtain. However, those axioms may in fact be assumptions which may or not may not hold in the given proof system of the theorem prover. By adding such axioms to the proof, the user inadvertently proves that a property ϕ holds only under the assumptions added to the proof, instead of proving that a property ϕ holds in general for some chosen domain (cf. [Pel01]). In contrast to the purist approach, the engineering-oriented approach allows tactics to be implemented as programs, e.g. Coq has its own tactic language Ltac (cf. [BC04]), or as external plugins, e.g., Isabelle/Hol (cf. [Nip02]). These tactics are applied to some subgoal in order to obtain another subgoal of the proof (cf. [Pel01]). Whatever the approach used for theorem proving, a proof is a result of the interaction between the theorem prover and the inputs provided by an expert (cf. [Pel01]), that has both knowledge of the theorem proving method provided in the theorem prover as well as the domain in which the proof is carried out.

Model Checking

Model checking is a computer-aided formal verification method, which allows the analysis of dynamical systems that can be modeled by state-transition systems (cf. [CHV18]). It can be used at design-time of an automated safety-critical system in order to verify the system with respect to formally specified system requirements. Figure 2.10 depicts the general process of model checking.

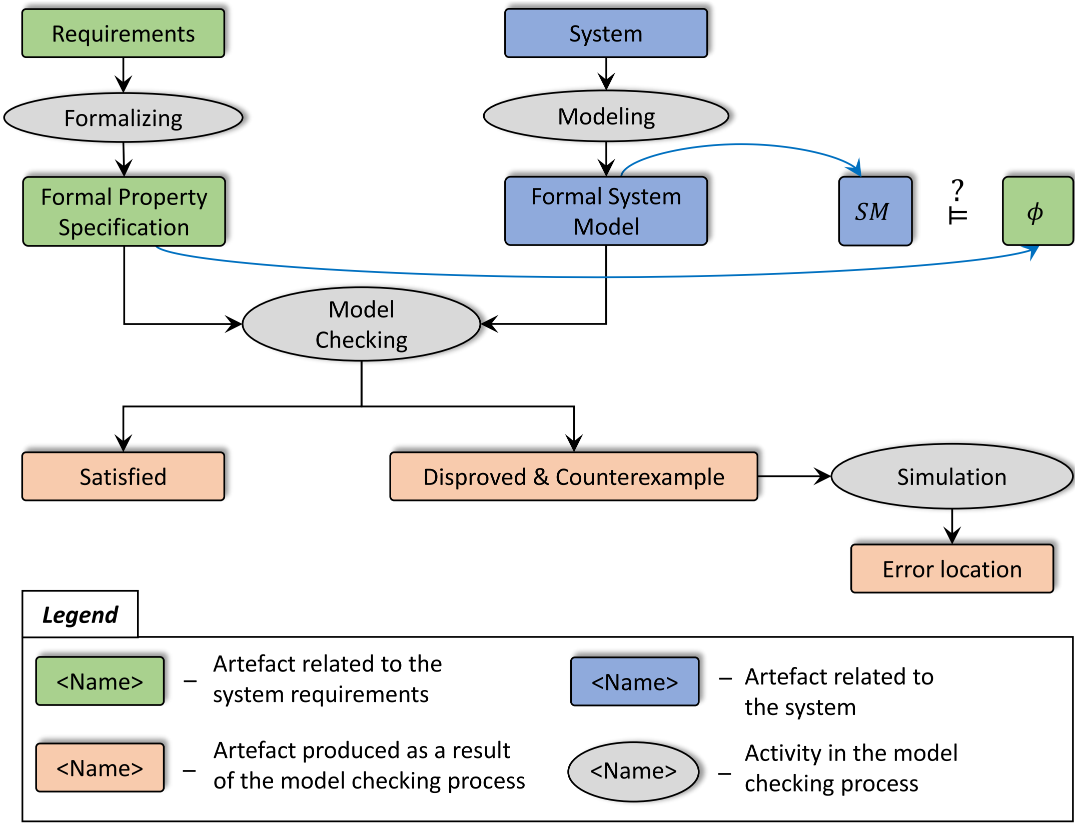

Requirements
Formalizing
Formal Property Specification
System
Modeling
Formal System Model
SM
⊨ ?
ϕ
Model Checking
Satisfied
Disproved & Counterexample
Simulation
Error location
Legend
<Name> – Artefact related to the system requirements
<Name> – Artefact related to the system
<Name> – Artefact produced as a result of the model checking process
<Name> – Activity in the model checking process

Figure 2.10.: Process of Model Checking (cf. [BK08]).

A model checker takes as input a formal system model, e.g., a finite state machine, and a formal property specification, e.g., expressed in temporal logic (cf. [BK08]). Given a model of the system under analysis SM and a property specification ϕ to verify, the model checker aims to answer the question whether the system model SM satisfies the property ϕ, denoted as $SM \models \phi$. For this purpose, it explores the state space of the system model exhaustively in search of states which disprove the given system property (cf. [AHDR18, AVR19a]). In the classical view of temporal logic model-checking, the state space of the system model is represented as a finite directed graph (cf. [CHV18]), which is explored with the help of graph traversal algorithms. During the model checking, the directed graph is rolled out in a tree-like structure. In this structure, the set $Reach^i$ denotes the states, i.e. the nodes, which are reachable within i transitions from the initial state of the system model. Figure 2.11 gives a visual intuition of how model checking works. The states which disprove the property to be verified are called henceforth *error states*. In case it has encountered an error state, the model checker returns a counterexample. For non-probabilistic systems and linear time safety properties, a counterexample is a single finite path, which starts in the initial state of the system model and ends in an error state (cf. [AVR19a, AVR19b]).

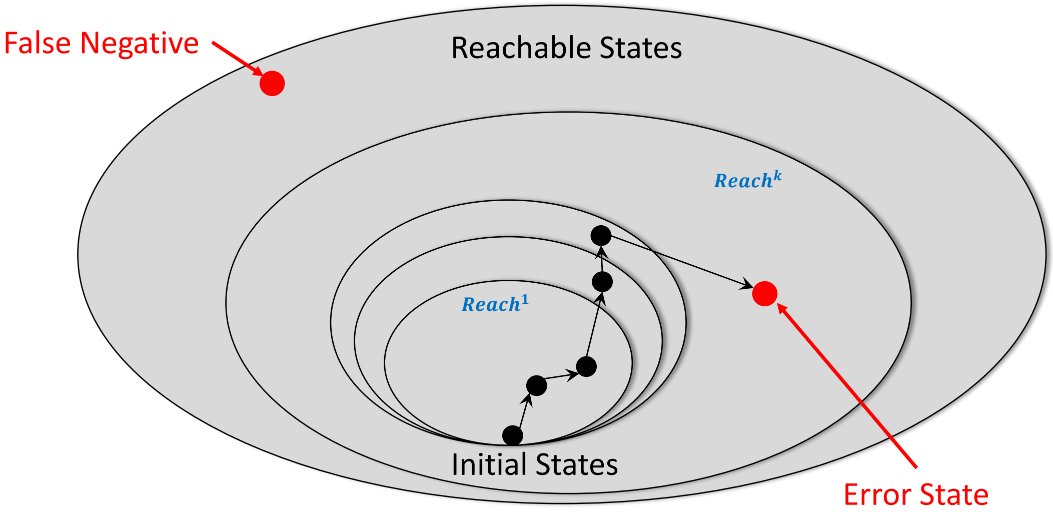


Figure 2.11.: Visual Intuition of Model Checking.

In probabilistic systems, there are two categories of properties which are considered for verification: quantitative properties and qualitative properties. Quantitative properties ask about the proportion of paths in a Markov model that satisfy a certain condition (cf. [BdAFK18]). Such properties express quantitative constraints with respect to the probability of occurrence of certain events, e.g., the probability for a message to be delivered within t seconds is at least 0.98 (cf. [BK08]) or the probability of a system failure occurring is at most 0.02 (cf. [FKNP11]). In turn, qualitative properties require typically that a good event will happen almost surely, i.e., with probability 1, or that a bad event almost never occurs, i.e., with probability 0 (cf. [BK08]). In probabilistic systems, qualitative properties are often expressed by formulae of the form $\mathbf{P}_{>0}(\psi)$ or $\mathbf{P}_{=1}(\psi)$

(cf. [BdAFK18]). Qualitative properties are considered a special case of quantitative properties with the probability bounds 0 and 1 (cf. [BK08]). The difference between qualitative properties and quantitative properties lies in the way their probability bounds are defined, i.e., for quantitative properties this is a rational number in the interval $[0, 1]$, while for qualitative properties it is either 0 or 1 (cf. [EKVY07]). Notice that safety and liveness properties as introduced in Definitions 2.3.1 and 2.3.2 are qualitative properties. Nevertheless, safety properties can also be expressed quantitatively, e.g., the probability of an error occurring is at most 0.01 (cf. [KNPQ10]).

In contrast to classical model checking, probabilistic model checking combines numerical methods with reachability analysis and standard model-checking techniques to show that the model of a probabilistic system satisfies a given property (cf. [ÁBD+14]). There is a wide range of formalisms that can be used to model probabilistic systems (cf. Section 2.4.2). Counterexamples obtained through probabilistic model checking are more complex than counterexamples generated by non-probabilistic model checking. Consider for this purpose the safety property ψ_1: "an error never occurs" in the context of a non-probabilistic system. The corresponding safety property in a probabilistic system is ψ_2: "an error occurs with probability at most p". When checked with a non-probabilistic model checker, the safety property ψ_1 can be refuted with a single finite path that starts in the initial state of the system model and ends in an error state (cf. Figure 2.11). In contrast to ψ_1, the safety property ψ_2 is refuted by a set of finite paths that reach the error state and whose total probability is larger than p (cf. [FKP10]). As a visual intuition, a counterexample in the probabilistic model checking is a tree starting in the initial state of the system model, whose leafs are all error states. This visual intuition is depicted in Figure 2.12.

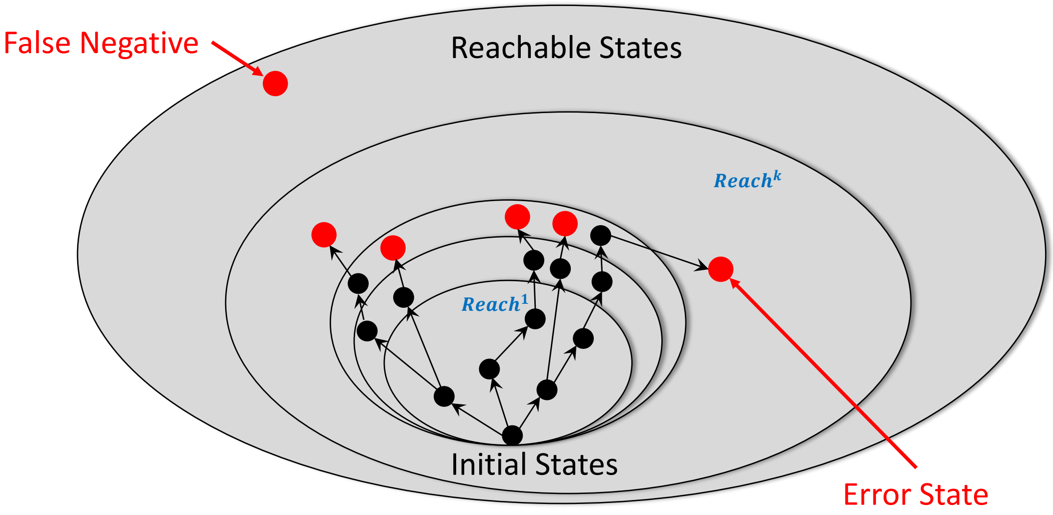


Figure 2.12.: Visual Intuition of Probabilistic Model Checking.

Ábraham et al. survey various methods by which counterexamples can be obtained for discrete-time Markov models (cf. [ÁBD+14]). There are three forms of counterexamples identified in [ÁBD+14]: (1) path-based counterexamples, (2) critical subsystems, and

(3) counterexamples based on the description language used to describe the system model. Path-based counterexamples are obtained by the explicit enumeration of the paths contained in the counterexample, usually starting with the most probable path and in the order of descending probability (cf. [ÁBD+14]). The path enumeration stops once the cumulative probability of the enumerated paths exceeds the probability bound specified in the verified property (cf. [ÁBD+14]). The path enumeration method has been applied to DTMCs (cf. [HK07], [WBB09]), and to MDPs (cf. [AL09]). Since the number of enumerated paths can become quite large, another form of obtaining counterexamples is to compute a critical subsystem of the system model under analysis. This is a sub-model of the system model at hand in which an error state is reached with a probability exceeding the specified probability threshold. The critical subsystem induces a counterexample by the set of its paths (cf. [ÁBD+14]). Critical subsystems can be extracted from DTMCs (cf. [Jan15]) and MDPs (cf. [WJÁ+14]). Rather than having counterexamples expressed in terms of states and paths of the system model, it is possible to describe a counterexample in terms of the modeling language used for the description of the system model. Typically, large Markov models are written in a human-readable language, which can prove an advantage for system designers that wish to inspect and analyze the obtained counterexamples (cf. [ÁBD+14]). Prism is a high-level modeling language in which a system model may consist of one or several models, whose behavior is described by guarded commands (cf. Section 2.4.3). In this case, a counterexample is a critical set of commands, which induces a Markov model that disproves the property to be verified. Counterexamples based on Prism as a high-level modeling language can be generated for PAs (cf. [WJV+13]).

The counterexamples obtained during model checking in case a property is disproved can serve as diagnostic information (cf. [BK08]). System designers can take this information and, based on it, refine or redesign the system model, followed by a rerun of the model checking procedure on the adapted system model and the property to be verified. This process is repeated until the system model satisfies the specified property. During the verification via a model checker, it may happen that the model checker runs out of memory. This is because the number of states in the system model exceeds the available amount of computer memory, which is denoted as the *state-space explosion problem* (cf. [BK08]). In fact, for model checking techniques, the cost in terms of memory usage may be exponential in the size of the system under verification (cf. [PGGB+08]).

In order to make the system model verifiable through model checking, system designers can try to reduce the model, e.g., restrict the value intervals of state variables. Modern model checkers employ various techniques in order to circumvent the state-space explosion problem. In order to cope with the infinite state space of timed automata, the Uppaal model checker uses the notion of zones, which are conjunctions of atomic clock constraints, and their representation as difference-bound matrices to obtain a coarser and more compact representation of the state space (cf. [BFL+18]). The model checkers Prism and Storm use among other methods a symbolic approach, in which the state space of the system model is encoded using binary decision diagrams (BDDs) as well as multi-terminal BDDs (MTBDDs) as data structures (cf. [BdAFK18]). Multi-terminal BDDs extend BDDs by allowing the representation of functions that map to numbers rather than just to boolean truth values, i.e. *true* or *false* (cf. [Hen18]). Despite effective methods used to address the state space explosion problem, models created for realistic systems may still be too large to fit into memory (cf. [BK08]) or for the verification procedure to terminate in a reasonable amount of time.

Compositional Verification

Besides symbolic approaches that aim at a more efficient representation of the system state space, a further method which addresses the state space explosion problem in order to ameliorate its effects is compositional reasoning or compositional verification.

Compositional verification has its roots in Hoare's calculus [Hoa69] and in the denotational semantics defined by Scott and Strachey [SS71] for computer languages, which establish compositional reasoning for sequential programs (cf. [GNP18]). Compositional reasoning for concurrent programs is introduced by Owicki and Gries (cf. [OG76]) as well as Lamport (cf. [Lam77]) and Abadi (cf. [AL93]). Pnueli and Harel introduce modular reasoning for reactive systems and studies the application of temporal logic to it (cf. [Pnu85], [HP85]). Fully compositional techniques for reasoning about network of processes and parallel programs are presented by Misra and Chandy (cf. [MC81]) and respectively by Jones (cf. [Jon83b], [Jon83a]). These methods impose an assumption on the inputs of each process and a guarantee on its outputs, providing also proof rules to ensure consistency between the assumptions and the guarantees of the programs under analysis (cf. [GNP18]). Therefore, these methods are denoted in literature as *assume-guarantee (A/G)* reasoning methods (cf. [GNP18]).

Assume-guarantee reasoning uses a divide-and-conquer approach for the verification of complex systems, in which individual components are analyzed separately (cf. [PGGB$^+$08]). This analysis is carried out based on assumptions made about the behavior of the other components in the system under analysis. Thus, rather than verifying a property of the whole system under analysis, A/G reasoning works on the premise that the system property can be decomposed into properties that are specific to the individual system components. Each component is then verified individually against its own property, under consideration of the knowledge about the context in which the component is supposed to operate correctly (cf. [PGGB$^+$08]). The context knowledge is encoded by assumptions that capture the requirements or the expectations which a system component has of the environment in which it operates (cf. [PGGB$^+$08]). The A/G paradigm works with formulae of the form: $\langle A \rangle\ M\ \langle G \rangle$ where M is a component in a given system, A is an assumption about M's environment, and G is a system property (cf. [CGP03]). The formula $\langle A \rangle\ M\ \langle G \rangle$ is considered to be true, if whenever the component M is part of a system that satisfies the assumption A, then the system must also satisfy the system property G (cf. [Pnu85], [PGGB$^+$08]).

Barringer et al. introduce various A/G proof rules for compositional verification (cf. [BGP03]). The asymmetric proof rule is considered to be useful in checking safety properties in a compositional manner (cf. [PGGB$^+$08]) and it is used in this section to give an intuition of how A/G reasoning works. Given a system that consists of two components M_1 and M_2, A/G reasoning says that if the formulae ψ_1 and ψ_2 hold, then the formula ψ also holds (cf. [CGP03].

$$\frac{\begin{array}{c}(\psi_1):\ \langle \mathit{true}\rangle\ M_1\ \langle A\rangle \\ (\psi_2):\ \langle A\rangle\ M_2\ \langle G\rangle\end{array}}{(\psi):\ \langle \mathit{true}\rangle\ M_1 \mid\mid M_2\ \langle G\rangle}\quad [\textsc{Asymm}]$$

Assumptions can be specified directly by domain expert, or they can be automatically learned or synthesized. There is extensive work done on learning-based assumption generation for non-probabilistic systems (cf. [CGP03], [GPB05], [PG06], [GGP07], [GBPG08], [PGGB$^+$08]). The method introduced by Păsăreanu et al. [PGGB$^+$08] uses L*, which is a learning algorithm first introduced by Angluin [Ang87] and then improved by Rivest and Shapire [RS93], to learn assumptions in an iterative manner. The algorithm L* takes an alphabet Σ as input and learns an unknown language $L_{Learned}$ over Σ. As a result, L* produces a deterministic finite state machine which accepts the language $L_{Learned}$ (cf. [CGP03], [PGGB$^+$08]). The L* algorithm works by interrogating a teacher, which answers two types of questions. The first type of question is a query of membership for a string $s \in \Sigma*$ in the language $L_{Learned}$. The second type of question is a conjecture, in which L* asks the teacher whether $\mathcal{L}(C) = L_{Learned}$ for a candidate finite state machine C built by L*, i.e., if the language accepted by C coincides with the learned language $L_{Learned}$ (cf. [PGGB$^+$08]). The teacher is in fact a model checker which, in case the languages $\mathcal{L}(C)$ and $L_{Learned}$ do not coincide, produces a counterexample that shows how the two languages differ from each other (cf. [PGGB$^+$08]).

The assumptions learned through this method are assumptions under which the system under analysis is shown to satisfy its safety property. Cobleigh et al. [CGP03] show how this works for the proof rule Asymm and non-probabilistic systems that are modeled as finite labeled transition systems (cf. [CGP03]). For the proof rule Asymm, the compositional verification of $M_1 \mid\mid M_2$ checks whether the formulae $\langle \mathit{true}\rangle\ M_1\ \langle A\rangle$ and $\langle A\rangle\ M_2\ \langle G\rangle$ hold in an iterative manner (cf. [CGP03]). At iteration i, the model checker verifies in the first step whether the component M_1 satisfies the property G under an assumption A_i. If the verification result is *true*, then the model checker proceeds with the second step, i.e. checking the component M_2 against the assumption A_i. Otherwise, the model checker returns a counterexample in the first step which shows that the assumption A_i is too weak for G to be satisfied (cf. [CGP03]). The assumption A_i is then strengthened using the information delivered by the counterexample of the first step, i.e. behaviors are removed from the assumption. The second step verifies the component M_2 against the assumption A_i. In case the result is *true*, then the the system $M_1 \mid\mid M_2$ satisfies the property G. Otherwise, the model checker gives out a counterexample which may illustrate that assumption A_i is to strong and it must be weakened in the next iteration step, i.e. behaviors are added to the assumption (cf. [CGP03]). The learning approach presented in [CGP03] learns assumptions only for asymmetric A/G rules in systems with two components. The approach has been generalized for systems with n components, where $n \geq 2$ (cf. [PGGB$^+$08]). Further extensions of this approach allow learning assumptions for symmetric A/G rules, i.e., the assumptions are learned for all components of a system under analysis simultaneously (cf. [BGP03]), and for circular A/G rules, i.e., the assumptions are learned for a system in which the first and the last component in the proof chain of the rule's premise coincide (cf. [PGGB$^+$08]).

In the context of probabilistic systems, Kwiatkowska et al. [KNPQ10] present an approach for compositional verification which builds upon the concepts in A/G reasoning from [CGP03], [GPB05] and [PGGB$^+$08]. The targeted system models are represented as PAs, which in this thesis are considered to be equivalent to MDPs (cf. Section 2.4.3). The respective assumptions and guarantees are probabilistic safety properties, which are represented as deterministic finite automata (cf. [KNPQ10]). The A/G triples on which probabilistic A/G reasoning works are of the form $\langle A\rangle_{\geq p_A}\ \mathcal{M}\ \langle G\rangle_{\geq p_G}$ where $\mathcal{M}$ is an MDP and $\langle A\rangle_{\geq p_A}$ and $\langle G\rangle_{\geq p_G}$ are probabilistic safety properties (cf. [KNPQ10]). Informally, this triple expresses that whenever the component $\mathcal{M}$ is part of system that satisfies the assumption A with probability at least p_A, then the system shall satisfy the property G with probability at least p_G (cf. [KNPQ10]).

For the asymmetric proof rule, the A/G semantics is defined as in [KNPQ10]. If $\langle A\rangle_{\geq p_A}$ and $\langle G\rangle_{\geq p_G}$ are two probabilistic safety properties each with their alphabets of actions Act_A and respectively Act_G, and $\mathcal{M}$ is an MDP with its alphabet of actions $Act_{\mathcal{M}}$, where $Act_G \subseteq Act_A \cup Act_{\mathcal{M}}$, then the following holds:

$$\langle A\rangle_{\geq p_A}\mathcal{M}\langle G\rangle_{\geq p_G} \Leftrightarrow \forall\, \mathcal{U} \in Sched_{\mathcal{M}[Act_A]}.(Pr^{\mathcal{U}}_{\mathcal{M}[Act_A]}\langle A\rangle_{\geq p_A} \rightarrow Pr^{\mathcal{U}}_{\mathcal{M}[Act_A]}\langle G\rangle_{\geq p_G})$$

This means that, the probabilistic triple $\langle A\rangle_{\geq p_A}\mathcal{M}\langle G\rangle_{\geq p_G}$ holds true, if and only if under all schedulers of $\mathcal{M}$, if $\mathcal{M}$ is part of a system that satisfies A with probability at least p_A, then the system will satisfy G with probability at least p_G.

Conversely, a triple $\langle A\rangle_{\geq p_A}\mathcal{M}\langle G\rangle_{\geq p_G}$ is false if and only if there exists a scheduler $\mathcal{U}$ which satisfies the assumption $\langle A\rangle_{\geq p_A}$ and violates the guarantee $\langle G\rangle_{\geq p_G}$ (cf. [KNPQ10]).

Notation 2.5.1. $M[Act_A]$ denotes the extension of the MDP $\mathcal{M}$ with the alphabet Act_A. This extension is obtained by adding to every state of $\mathcal{M}$ a self-loop labeled with an action a, for each $a \in Act_A \setminus Act_{\mathcal{M}}$ (cf. [FKP10]). ■

Kwiatkowska et al. [KNPQ10] use the asymmetric proof rule to exemplify how probabilistic A/G reasoning works. Given two MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$ and two probabilistic safety properties $\langle A\rangle_{\geq p_A}$ and $\langle G\rangle_{\geq p_G}$, with $Act_A \subseteq Act_{\mathcal{M}_1}$ and $Act_G \subseteq Act_A \cup Act_{\mathcal{M}_2}$, probabilistic A/G reasoning says that if the formulae ψ_1 and ψ_2 hold, then the formula ψ also holds (cf. [KNPQ10]).

$$\frac{\begin{array}{c}(\psi_1):\ \langle true\rangle\ \mathcal{M}_1\ \langle A\rangle_{\geq p_A}\\ (\psi_2):\ \langle A\rangle_{\geq p_A}\ \mathcal{M}_2\ \langle G\rangle_{\geq p_G}\end{array}}{(\psi):\ \langle true\rangle\ \mathcal{M}_1\ ||\ \mathcal{M}_2\ \langle G\rangle_{\geq p_G}}\quad [\textsc{Asymm-Prob}]$$

The verification of a probabilistic A/G triple $\langle A\rangle_{\geq p_A}\ M\ \langle G\rangle_{\geq p_G}$ is reduced to the problem of multi-objective model checking (cf. [KNPQ10]). Multi-objective model checking is applied to the product between the MDP $\mathcal{M}$ and the deterministic finite automata that correspond to the assumption $\langle A\rangle_{\geq p_A}$ and respectively to the guarantee $\langle G\rangle_{\geq p_G}$ (cf. [KNPQ10]).

Multi-objective model checking is introduced in [EKVY07], as an approach which verifies MDPs with respect to multiple linear-time properties. Etessami et al. [EKVY07] investigate model checking of MDPs with respect to both qualitative as well as quantitative multi-objective queries. In general, the problem of multi-objective model checking is formulated as follows. Given an MDP $\mathcal{M}$, a set of linear-time properties ϕ_i, and a set of probabilities $p_i \in [0, 1]$, with $i = \{1, \ldots, k\}$, the goal is to find a strategy or a scheduler $\mathcal{U}$ such that, for any i, the property ϕ_i is satisfied with a probability of at least p_i by a path in $\mathcal{M}$ produced by the scheduler $\mathcal{U}$ (cf. [EKVY07]). Formally it must be shown that $\exists\, \mathcal{U} \in \mathit{Sched}_{\mathcal{M}}$ so that $\bigwedge_{i=1}^{k}(\mathit{Pr}_{\mathcal{M}}^{\mathcal{U}}(\phi_i) \bowtie p_i)$ holds, where $\bowtie \in \{\geq, >\}$ (cf. [KNPQ10]).

Notation 2.5.2. Given an MDP $\mathcal{M}$ and a scheduler $\mathcal{U} \in \mathit{Sched}_{\mathcal{M}}$, then

$$\mathit{Pr}_{\mathcal{M}}^{\mathcal{U}}(\phi_i) \overset{\text{def}}{=} \mathit{Pr}_{\mathcal{M}}^{\mathcal{U}}(\{\pi \in \mathit{Paths}_{\mathcal{M}}^{\mathcal{U}} \mid \pi \models \phi_i\})$$

where:

- $\mathit{Paths}_{\mathcal{M}}^{\mathcal{U}}$ denotes the set of all paths through $\mathcal{M}$ when controlled by the scheduler $\mathcal{U}$ (cf. [KNPQ10]).
- $\mathit{Pr}_{\mathcal{M}}^{\mathcal{U}}(\phi_i)$ denotes the probability that the property ϕ_i is satisfied by all paths in the MDP $\mathcal{M}$ induced by the scheduler $\mathcal{U}$ (cf. [KNPQ10]).
- $\mathit{Pr}_{\mathcal{M}}^{\mathcal{U}}(\{\pi \in \mathit{Paths}_{\mathcal{M}}^{\mathcal{U}} \mid \pi \models_{\mathit{Sched}_{\mathcal{M}}} \phi_i\})$ denotes probability that all infinite paths π in $\mathcal{M}$ induced by the scheduler $\mathcal{U}$ satisfy the property ϕ_i (cf. [KNPQ10]).

■

Besides the asymmetric rule, Kwiatkowska et al. formulate and prove further rules for probabilistic A/G reasoning in [KNPQ10]. Thus, two generalizations of the asymmetric rule are formulated: the first generalization shows how the rule can be applied to a system with more than two components, while the second generalization extends the asymmetric proof rule to k assumptions, with $k > 1$ (cf. [KNPQ10]). Other A/G rules presented in [KNPQ10] are an asynchronous proof rule which accounts for systems with asynchronous components, and a circular proof rule. The A/G reasoning framework in [KNPQ10] is extended in [KNPQ13] with compositional verification techniques and A/G proof rules for a more general class of quantitative properties, which includes probabilistic ω-regular properties, e.g, probabilistic LTL properties and probabilistic safety properties, as well as expected total cost or reward properties. The work in [KNPQ13] introduces numerical queries in the context of A/G reasoning, in order to compute minimum and maximum bounds for the probability with which a PA satisfies a given property.

Along the A/G reasoning techniques as they are briefly presented in this section, there are various other methods for compositional verification. For example, Benveniste introduces A/G contracts for non-probabilistic systems in [BCP07] and Delahaye extends them with probabilities in [Del10] and [DCL11]. Assume-guarantee contracts fall in the A/G reasoning spectrum as the techniques presented in this section. Another compositional verification method, compositional reachability analysis builds the global state machine of system under analysis from its component processes in stages, based on the specified system hierarchy (cf. [CK95]). Contextual reachability analysis includes context constraints in compositional reachability analysis, in order to minimize the built state machine with respect to system properties of interest (cf. [CK96]). Interface automata are an automata-based language which captures in the same model both input assumptions about the order in which the methods of a component are called, and output guarantees about the order in which the component invokes methods of other components in the system (cf. [dAH01]).

2.5.3. Runtime Verification

When applied to an automated safety-critical system, design-time verification methods work on the basis of an abstract representation of the system under analysis rather than on the system itself, e.g., a finite state-transition model in model checking or a set of theories in theorem proving. The abstract representation reflects the most relevant aspects of the system under analysis, and checking it for correctness with respect to a given property specification provides useful insights about the system itself.

When an automated safety-critical system is deployed in its operational environment, it may behave slightly different than the technical system model verified during design-time. One reason may be that some information is only available at runtime. Furthermore, the environment in which the system is deployed may be very different from the environment model built at design-time and used during the design-time verification process.

Runtime verification is considered to be a lightweight verification method, which is complementary to design-time verification as well as to testing (cf. [LS09]), as it is primarily performed at runtime. Bartocci et al. point out that runtime verification is known in the scientific community also under various other names, e.g., runtime monitoring, trace analysis, or dynamic analysis (cf. [BFFR18]). Some scholars consider runtime monitoring to be a specific form of verification (cf. [BFFR18]). This is because due to its name, runtime monitoring conveys the idea of an interaction which takes place between the system under analysis and the monitor, while verification is regarded in general as a more passive approach (cf. [BFFR18]). The notion of runtime verification as it is understood in this thesis is given in Definition 2.5.13.

Definition 2.5.13 - Runtime Verification
Runtime verification is an activity in which verification techniques are used during the runtime of a system under analysis in order to check whether a run of the system satisfies or violates a given correctness property (cf. [LS09]). ■

Notice that a run of a system under analysis is considered to be an infinite sequence of system states or system actions (cf. [LS09]), and thus, similar to a trace of the system (cf. Definition 2.5.7). Since the current system execution is a finite trace of a system (cf. Definition 2.5.8), it can be said that the system execution is a finite, though continuously extended, prefix of the corresponding evolving system run (cf. [LS09]). In comparison to model checking, which asks whether a system trace or more general all system traces fulfill a specified correctness property, runtime verification analyzes primarily system executions (cf. [LS09]).

In order to verify whether an execution satisfies a given correctness property, runtime verification uses monitors. A monitor can be regarded as a decision procedure which decides whether the current system execution fulfills the correctness property and returns a truth value *true/false* or *yes/no* (cf. [LS09]). The notion of runtime monitor as it is understood in this thesis is introduced in Definition 2.5.14.

Definition 2.5.14 - Runtime Monitor
A *runtime monitor* is a computational entity, i.e. device or a decision procedure, which executes in parallel to a system under analysis and observes its runtime behavior (cf. [BFFR18]). When sufficient observations of the system behavior are gathered, the monitor evaluates the observed system execution with respect to a given property and yields a certain verdict (cf. [LS09], [BFFR18]). ■

Observations of System Behavior

The notion of system behavior is connected to the way a system under analysis changes over time, e.g., through an update of its internal state or by carrying out some action which may affect its environment in a certain way (cf. [BFFR18]). Bartocci et al. describe the behavior of a system under analysis in terms of the observation that can be made about it (cf. [BFFR18]). Observations can be made by inspecting the system state at particular steps in the system execution, or by recording state changes or actions (cf. [BFFR18]) which are carried out by the system and reflect the system evolution throughout its execution. Observations made about a system under analysis are also denoted as events (cf. [HR17], [BFFR18]). An event is something that can happen both in the system under analysis and in its environment (cf. [BFFR18]) and can be represented as a data record received by the runtime monitor (cf. [HR17]). In this thesis, the data records that pertain to an event are considered to be an interpretation of the state variables. As such, runtime observations are considered in terms of system states.

System Property and Property Specification

The notion of property is related to the way some behavior of a system under analysis is described. Bartocci et al. differentiate between the notion of property and the notion of specification, with a property being a possibly infinite set of traces and a specification being a (textual) artifact which describes a property, and therefore describes the set of traces (cf. [BFFR18]). A specification is related to a concrete specification language, while a property is an artifact which is unique and independent of a specification language, that describes some behavior of a system under analysis (cf. [BFFR18]). Therefore a property may have one or more specifications (cf. [BFFR18]). In this thesis, the notion of *property* is considered to be related to the notion of *requirement.* In general, a property or a requirement is considered to be a statement about a system under analysis or some aspect of it. It can express a desired or an undesired behavior of the system. In this thesis, the notion of property or requirement refers primarily to textual requirements written in informal language, e.g., in English, and which are managed in requirements catalogs using dedicated tools, e.g., IBM DOORS (cf. [AHDR18]). In turn, the notion of *property specification* denotes the (textual) specification of the property in question, written in a formal language.

Formal Specification Languages for Runtime Verification

Formal specification languages for runtime verification can be assigned to two categories: executable or declarative (cf. [BFFR18]). When expressed in an executable language, e.g., as a state machine, a property specification is directly executable. In turn, declarative languages, e.g., temporal logic, are used to formulate the property specification, from which an executable monitor is then generated. Executable specifications tend to be more at an operational level and less practical for capturing properties at a high-level of abstraction. However, executable specification may also have more straightforward monitoring algorithms (cf. [BFFR18]).

Bartocci et al. [BFFR18] review the families of specification languages used in runtime verification. One of the most common temporal logics used in runtime verification is linear temporal logic (LTL) [Pnu77]. There are two forms of LTL: future-time LTL and past-time LTL. Future-time LTL is the classical LTL, which has has two basic modal operators: X (*Next*) and U (*Until*), with the help of which it can be reasoned about system executions in the future. These operators are used to define two further modal operators: G (*Globally*) and F (*Eventually*) (cf. [BFFR18]). Past-time LTL is a variant of LTL which reasons about system executions situated in the past, using two modal operators which are symmetric to the operators X and U: *Previous* as the dual of the X operator and *Since* as the dual of the U operator (cf. [BFFR18]).

Over the years, LTL has received various extensions. One of these extensions is interval temporal logic (cf. [CZ97], [ZZC05]). Rather than reasoning about discrete events or states, interval temporal logic reasons over intervals, which are pairs of start and end points. Formulas in interval temporal logic use binary relations to compare intervals, e.g., check whether intervals overlap or are included in one another (cf. [BFFR18]).

Other extensions of LTL increase its expressiveness in order to allow the formulation more complex properties. Frequency linear temporal logic annotates the usual U operator of LTL with a rational number $c \in \mathbb{Q} \cap [0, 1]$, e.g., $\phi\ U^{0.5}\ \psi$ means that ϕ should hold with the frequency 0.5 until ψ holds (cf. [BDL12]).

Formal specification languages belonging to the LTL family of temporal logics have a qualitative view of time, i.e. they can express an ordering of events, but cannot relate these events with the quantitative timeline in which these events occur (cf. [BFFR18]). There are formal temporal logics which extend LTL with the notion of quantitative time by annotating the underlying system trace with timestamps, e.g. metric temporal logic (MTL) [TR05] with its fragment metric interval temporal logic (MITL) as well as timed LTL (TLTL) [BLS11]. MTL annotates modal operators with discrete time intervals, e.g., $\phi\ U_{[5,9]}\ \psi$ means that ψ holds at some point between 5 and 9 time units and ϕ holds in every time instant before that. Furthermore, MTL uses the notion of congruence in order to require that a formula holds periodically with respect to an absolute time (cf. [BFFR18]). TLTL formulas can indicate that the time since the last occurrence of a given event or until the next occurrence of the event is situated within a specific interval I (cf. [BFFR18]). The formal logics which provide a quantitative notion of time are used to reason about real-time systems.

Signal temporal logic (STL) [MN04] and its different variations have been introduced in order to reason about cyber-physical systems, which are a combination of digital and physical systems. The dynamic behavior of digital systems can be described by state-transition formalisms, e.g. automata, which produce discrete sequences of states or of actions, while physical systems are modeled with differential equations, that output executions in form of continuous signals and trajectories (cf. [BDD$^+$18]). STL extends MTL with numerical predicates over real-valued variables (cf. [BDD$^+$18]). In STL, a trace is not a discrete sequence of system states or system actions anymore, but a collection of signals. A signal is defined as a function from a set of real-time points to a value domain (cf. [BFFR18]). Numerical predicates over signals can be used to define operators to capture different aspects of signals, e.g., the rising and falling edges of a signal (cf. [MN04]). STL provides a dense interpretation of time, which means that there is no X operator to reason about the next state, because a next state cannot be uniquely defined (cf. [BFFR18]).

STL has been extended in the last years beyond temporal properties and the new extensions of STL offer the possibility to reason about other types of properties, e.g, topological spatial requirements, e.g., the Signal Spatio-Temporal Logic (SSTL) presented in [BBM$^+$15] and [NBC$^+$15], the Spatial-Temporal Logic (SpaTeL) introduced in [HJK$^+$15], or the Spatio-Temporal Reach and Escape Logic (STREL) in [BBLN17].

Notice that this short presentation of formal languages used for property specification in runtime verification is not meant to be an exhaustive review of research works in this area. There are many more languages that go beyond the presented temporal logics and their variants, e.g., regular expressions (cf. [AAC$^+$05]), rule systems (cf. [BRH10], [Hav15]), languages that combine regular expressions with temporal logic (cf. [LS07]), and many more (cf. [HR17], [BFFR18]).

In this thesis, LTL is used as a formal language to express the property specification from which the runtime monitors are built.

Classification of Runtime Monitors

There are various types of runtime monitors, depending on when the monitoring of the system occurs and how the runtime monitor is deployed and executed in relation of the monitored system. With respect to the time when the monitoring is performed, there are two categories of runtime monitoring: offline monitoring and online monitoring.

Offline monitoring, also called logging, is an analysis which is carried out on recorded system executions. The execution of the monitor is independent of the execution of the system, and thus, constraints with respect to the runtime overhead created by a runtime monitor do not apply. Furthermore, since it has access to complete system executions, offline monitoring can be used to perform global analysis over the recorded system executions (cf. [BFFR18]). One big disadvantage of offline monitoring is that any violation of the property specification is detected after the system execution terminates. This is especially critical for automated critical-systems, as it reduces the chances to apply appropriate measures to mitigate the effects of the hazardous event that led to the property violation.

Online monitoring is the opposite of offline monitoring, as it is performed during the system's execution. It does not have the disadvantage of late detection or property violation as is the case for offline monitoring. Online monitoring operates however with partial system executions, i.e. up to the current execution step (cf. [BFFR18]). Furthermore, there are low overhead constraints for online monitoring. Runtime monitors are often deployed along their monitored systems and thus may share resources with one another, e.g. computation time, memory or bus bandwidth (cf. [BFFR18]). Therefore, it is important that monitor complexity with respect to the memory and computation requirements is as low as possible, so as not alter the monitored system in terms of its memory usage and response time and thus affect its functional and non-functional behavior (cf. [BLS11]).

There are two kinds of online monitoring, with respect to how a runtime monitor is executed in relation of the monitored system: synchronous online monitoring and asynchronous online monitoring. When the online monitoring is performed in a synchronous manner, the monitor and the system execute in lock-step (cf. [BFFR18]). Each time it performs an action or generates an event, the monitored system waits for the runtime monitor to process it before it carries on with the system execution (cf. [BFFR18]). In asynchronous online monitoring, the runtime monitor's execution is decoupled from the system's execution (cf. [BFFR18]). The monitor evaluates the system actions at its own pace and independently from the system (cf. [CFAI17]). In comparison to synchronous monitoring, asynchronous monitoring typically yields lower overheads and it is less intrusive in the system. Late detection of property violation can occur, especially if the monitored system and the runtime monitor are deployed on an platform which does not guarantee fairness between the executions of the both (cf. [BFFR18]). Notice that an asynchronous monitor still executes alongside the monitored system and evaluates finite system traces in an incremental fashion. Therefore asynchronous monitors are not to be confused with offline monitors which execute after the system execution is terminated and analyze prerecorded system traces (cf. [CFAI17]).

In order for runtime monitors to have knowledge of the actions performed by the monitored system, the system must be instrumented. Instrumentation denotes a mechanism used for the extraction of information from the monitored system, e.g., signals, events or system actions, as well as other information of interest (cf. [BFFR18]). Instrumentation can be applied both to hardware and software systems for the purpose of online monitoring. Cassar et al. [CFAI17] compare various instrumentation techniques and the monitoring methodologies and tools in which these techniques have been implemented.

Runtime Verification vs. Testing

Runtime verification bears similarities with testing, since it does not consider each possible execution of the system, but only one execution or a finite set of system executions (cf. [LS09]). Testing is usually carried out by stimulating the test object with a set of test inputs and checking whether the produced output corresponds to the one expected by the test engineers. For the activity of testing, test cases are defined by test engineers as finite sequences of system inputs and system outputs (cf. Section 2.5.1).

Runtime verification is closer to oracle-based testing (cf. [LS09], [BLS11]). In oracle-based testing, test cases are comprised of test inputs, a precondition, expected result and postcondition. The expected result and the postcondition are also denoted as the oracle information and respectively the oracle procedure (cf. Section 2.5.1). The test oracle observes the system execution with the test inputs provided in the test cases and compares the outputs emitted by the test object with the expected output specified by the oracle (cf. Definition 2.5.9). In terms of runtime verification, a test oracle acts as a runtime monitor (cf. [BLS11]).

The difference between oracle-based testing and runtime verification lies in the way the oracle and respectively the runtime monitor are obtained. While the test oracle is usually defined directly by systems engineers, the runtime monitor is often built manually or automatically synthesized from a property specification written in a formal language (cf. [LS09]). However, this line of demarcation between oracle-based testing and runtime verification begins to blur, due to the usage of model checking to automatically generate test case from formally specified requirements (cf. Section 2.5.1). In fact, test oracles can be realized through runtime monitors (cf. [KFK14]). There is however a more obvious difference between testing and runtime verification. In contrast to testing, runtime verification rarely aims to exhaustively test a system under analysis (cf. [LS09]).

2.6. Summary

The goal of this chapter was to present several fundamental concepts and approaches on which this thesis is based. The first section of this chapter introduced the notions of safety-critical system, functional safety and safety of the intended functionality. This section explained also the concepts of automated system and autonomous system, with a presentation of the degrees of automation/autonomy recorded in the literature. The first section then defined the notion of uncertain environment and highlighted its characteristics.

The second section of this chapter discussed the system life cycle and the development process proposed in the international standards ISO 26262 and ISO/PAS 21448, highlighting the artifacts produced in the development process that contribute to engineering the safety of automated safety-critical systems. The presentation of the system development process is followed in the third section of this chapter by a discussion of the specification and modeling formalisms used in this thesis for the formal specification of system requirements and modeling the behavior of automated safety-critical systems. The third section of this chapter introduces two formal logics, TCTL and PCTL, for specification of the system properties. In the forth section of this chapter, the modeling formalisms timed automata and MDPs, are introduced for modeling of automated safety-critical systems.

The final section of this chapter gives an overview of the methods used for the verification and validation of automated safety-critical systems with respect to specific system properties. Thus, this section presents briefly the theoretical foundations of testing, of various approaches for design-time verification and of runtime verification.

Chapter 3.

Problem Analysis

3.1. Motivational Example: Mobile Service Robot 76
3.2. Overall Development Process 81
3.3. Requirements Elicitation and Analysis 83
3.3.1. Informal Specification of System Requirements 83
3.3.2. Formal Specification of System Requirements 88
3.4. Safety Analysis 95
3.5. System Design 100
3.5.1. Usage of Formal Models in System Design 101
3.5.2. Environment Model 102
3.5.3. Technical System Model 104
3.5.4. Overall System Model 104
3.5.5. Specification of System Properties 108
3.5.6. Design-Time Verification 109
3.6. System Implementation 112
3.7. System Test 113
3.8. Requirements Validation 118
3.9. Analysis of Emerging Challenges 118
3.9.1. Challenges of Design-Time Verification 119
3.9.2. Challenges of Testing 120
3.10. Scope of this Thesis 123
3.10.1. Introduction of Runtime Monitoring of Environment Assumptions 124
3.10.2. Research Questions of this Work 125
3.11. Summary 126

The goal of this chapter is to carry out a problem analysis and identify the challenges that need to be addressed in this work. In order to achieve this goal, this chapter produces the following artifact as output:

Research Questions. The research questions formulate in a clear manner the problems addressed in this thesis. Once defined, the research questions will serve as a point of reference for the remainder of this work.

In order to create the research questions, a discussion on the challenges of this work must be performed. The basis of this discussion is constituted by a small motivational example, which is introduced in Section 3.1. Section 3.2 gives an overview of the development process used for the design and development of the example system. The following sections take a more in depth look at each phase of the development process: from requirements analysis (Section 3.3) and safety analysis (3.4), through system design (Section 3.5) and system implementation (Section 3.6), up to system test (Section 3.7) and requirements validation (Section 3.8). Several issues arisen during the design-time verification and system testing are identified and discussed in Section 3.9. As a consequence of these problems, the approach of this thesis is briefly introduced in Section 3.10. Subsequently, the challenges that emerge from it are analyzed and the research questions of this work are defined. In Section 3.11, a summary of the main ideas presented in this chapter is given.

3.1. Motivational Example: Mobile Service Robot

This section gives a high-level description of the example system used as basis for the problem analysis of this thesis. The system in question is a mobile service robot moving autonomously in an uncertain environment. An abbreviated version of this example has already been introduced in a previous paper [AAHR16]. Nevertheless, for the sake of completeness of the problem analysis, details of the example system presented in [AAHR16] are reiterated in this section.

A physical overview of the mobile service robot, and its environment is depicted in Figure 3.1. In this schematic display, the robot drives to its goal, while two obstacles move from the opposite direction on the same lane as the robot. The environment features two other obstacles, one stationary and one dynamic, which occupy the neighboring left and right lanes. The robot is equipped with sensors, which allow it to observe the changes in the environment in real time. The robot's sensors are configured so that the field of view spans up to a finite horizon h_R, represented by the grey area in front of the robot in Figure 3.1. Notice that, the fitting of the sensors on the robot allows for the robot to observe only the environment changes which occur in front of it. Due to these sensory limitations, the robot has only partial knowledge of its environment.

The environment is represented as a subset of the two-dimensional space spanned by the Cartesian coordinate system. In order to simplify the analysis, several considerations are made with respect to the physical environment and to the robot and the obstacles which move in this environment. Firstly, the robot and the obstacles in its environment are considered to be rigid objects. Notice that in physics, a rigid object is a solid object in which the deformation caused by external forces applied on the object is so small that it can be neglected (cf. [RP13]). In order to simplify the analysis on the motion of rigid objects, each kinematic quantity which describes the motion of a rigid object, e.g., velocity, acceleration, can be expressed in relation to one of the particles of the object, chosen as a reference point (cf. [RP13]). Typically, this point is chosen to be the center of mass of the rigid object. In this way, it is possible to approximate the robot

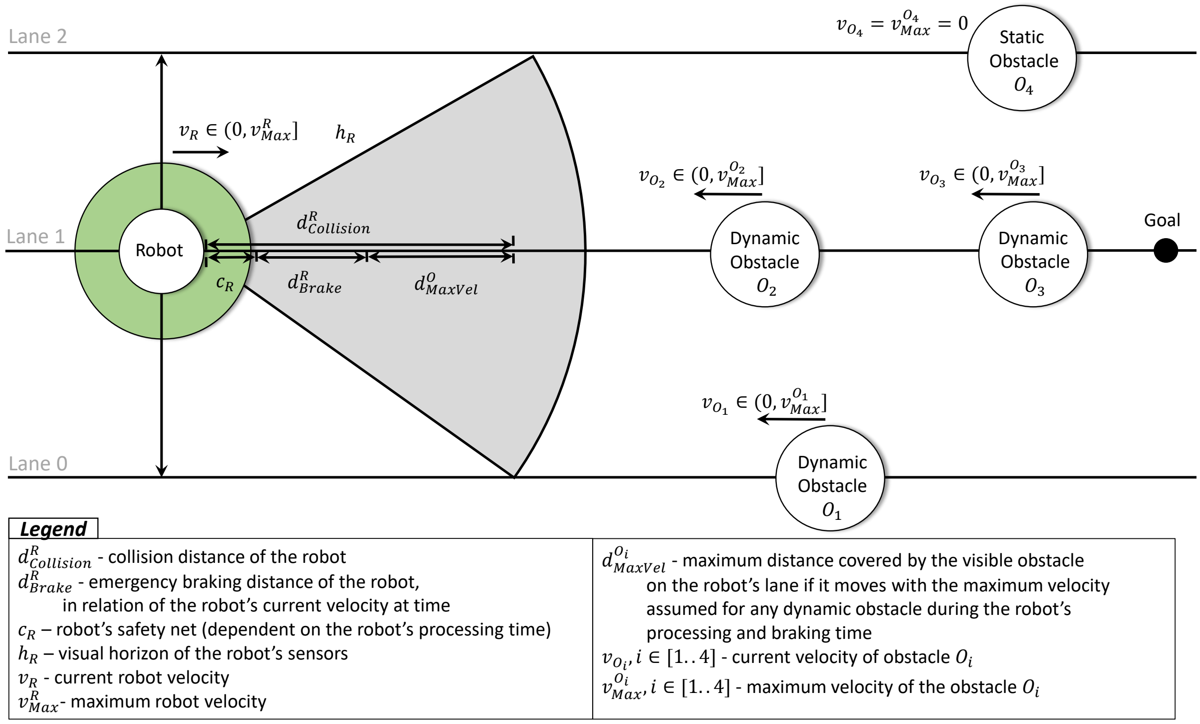

Lane 2
$v_{O_4} = v_{Max}^{O_4} = 0$
Static Obstacle O_4
$v_R \in (0, v_{Max}^R]$
h_R
$v_{O_2} \in (0, v_{Max}^{O_2}]$
$v_{O_3} \in (0, v_{Max}^{O_3}]$
Goal
Lane 1
Robot
$d_{Collision}^R$
c_R
d_{Brake}^R
d_{MaxVel}^O
Dynamic Obstacle O_2
Dynamic Obstacle O_3
$v_{O_1} \in (0, v_{Max}^{O_1}]$
Lane 0
Dynamic Obstacle O_1
Legend
$d_{Collision}^R$ - collision distance of the robot
d_{Brake}^R - emergency braking distance of the robot, in relation of the robot's current velocity at time
c_R – robot's safety net (dependent on the robot's processing time)
h_R – visual horizon of the robot's sensors
v_R - current robot velocity
v_{Max}^R- maximum robot velocity
$d_{MaxVel}^{O_i}$ - maximum distance covered by the visible obstacle on the robot's lane if it moves with the maximum velocity assumed for any dynamic obstacle during the robot's processing and braking time
$v_{O_i}, i \in [1..4]$ - current velocity of obstacle O_i
$v_{Max}^{O_i}, i \in [1..4]$ - maximum velocity of the obstacle O_i

Figure 3.1.: Mobile Service Robot: Physical Overview of the Motivational Example.

and the obstacles in its environment to discrete points in the two-dimensional space. The notion of object is henceforth interchangeable with that of robot and obstacle in robot's environment, even if the obstacle is a human being. Secondly, several physical characteristics of the lanes on which the robot and the obstacles move in the physical environment are abstracted from, e.g., lane width, lane margins, and lane curvature. Thus, the lanes are considered to be straight lines in the Cartesian plane.

Any object in this environment, including the robot, has a positive velocity limited by a specific maximum value, which is considered as a physical upper bound for the current velocity of the object. Before describing further the physical overview of the motivational example, several notations must be introduced. These notations are considered to be effective for the remainder of this work.

Notation 3.1.1. The coordinates of an object, be that the robot itself or an obstacle, are represented by the lane and the position on the lane where the respective object is situated. The lane is denoted by the variable y, while the position on the lane is represented by the variable x. The current velocity of an object is expressed by the variable v, while its specific maximum velocity is denoted by v_{Max}.

Any variable that refers to a feature of the robot contains the subscript or respectively the superscript R. Conversely, to denote that a variable refers to a feature of an obstacle, the subscript or respectively the superscript O is used. Thus,

1. (x_R, y_R) are the coordinates of the robot,
2. v_R and v_{Max}^R are the current velocity and the specific maximum velocity of the robot,

3. (x_O, y_O) are the coordinates of the obstacle O,
4. v_O and v^O_{Max} are the current velocity and the specific maximum velocity of the obstacle O, and
5. (x^R_{Goal}, y^R_{Goal}) are the coordinates of the robot's goal position or destination.

■

The behavior of the robot is kept simple. As it starts to drive, the robot accelerates until it reaches its maximum velocity. It continues to drive with this velocity until it reaches its destination or until it is forced to brake due to collision danger.

Whenever the robot detects a collision danger, there is a certain amount of time which passes by from the moment when the robot receives the sensor data until the moment when it acts upon this data. This time is henceforth denoted as the robot's *reaction time* and it consists of three main components: (1) the time to record the sensor data and send it to the robot's main processing unit, (2) the time it takes the robot's processing unit to process the sensor data and compute the necessary action, and (3) the time to send the action command to the robot's motor controller. Notice that recording and processing the sensor data, computing the necessary action, and sending the action command to the actuators are activities which depend heavily on the computational power of the robot's hardware and software platform. The processing of the sensor data consists in this example of the computation of the collision distance performed by a primitive collision avoidance mechanism with which the robot is equipped. During its reaction time, the robot continues to drive with its current velocity v_R for a certain distance c_R, as shown in Equation (3.1):

$$c_R = v_R * t^R_{Reaction} \tag{3.1}$$

$$t^R_{Reaction} = t^R_{Sensor} + t^R_{Processing} + t^R_{Actuator} \tag{3.2}$$

where:

- v_R denotes the robot's current velocity, and
- $t^R_{Reaction}$ is the reaction time of the robot accumulated from three components:
  - t^R_{Sensor}, which is the time necessary to record the sensor data,
  - $t^R_{Processing}$, which is the time needed to process the sensor data and compute a corresponding action, and
  - $t^R_{Actuator}$, which represents the time necessary to send the action command to the robot's actuators.

In order to ensure that the robot does not actively collide with an obstacle during the processing of its sensor data, a region as wide as the distance c_R is established around the robot and is factored in the computation of its collision distance. This is the green region around the robot in Figure 3.1 and it is henceforth denoted as the robot's *safety net.*

If a static or dynamic obstacle is situated inside the area spanned by the collision distance of the robot, then the robot perceives the obstacle as a collision danger and initiates collision avoidance maneuvers. There are two types of maneuvers which the robot employs in case of collision danger: (1) brake to a standstill and wait for the moving obstacle to pass by and (2) change to a safe lane.

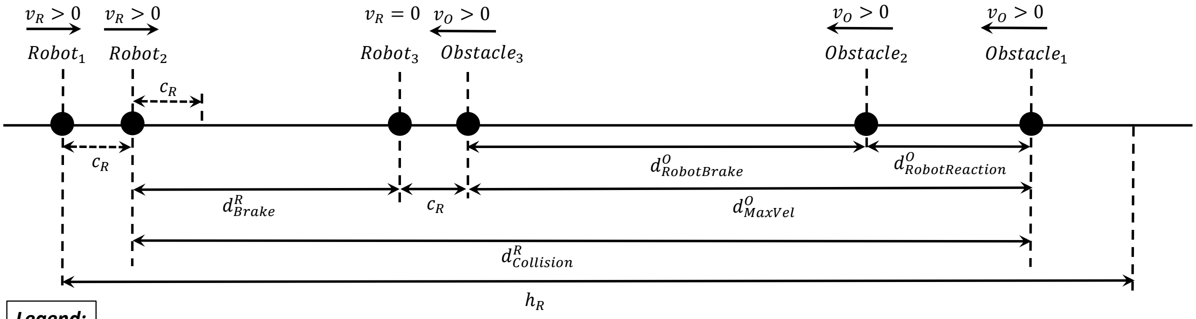
In this example, the main task of the robot during its processing step is the computation of the collision distance $d^R_{Collision}$ based on the data received from its sensors. Notice that this means that any computation of the collision distance is performed in relation to the visible obstacles, stationary or dynamic, in its environment. The collision distance between the robot and a moving obstacle is calculated as in Equation (3.3):

$$d^R_{Collision} = c_R + d^R_{Brake} + d^O_{MaxVel} \tag{3.3}$$

where:

- c_R is the safety net of the robot, defined as a function of the robot's current velocity v_R,
- d^R_{Brake} denotes the current braking distance of the robot, in relation of the robot's current velocity v_R, and
- d^O_{MaxVel} is the distance travelled by a visibile obstacle O, during the robot's reaction and braking time, while moving with the maximum velocity assumed for any dynamic obstacle in the robot's environment.

In order to illustrate the role of the robot's reaction time, and by extension that of its safety net, in the computation of the collision distance, a simple scenario involving the robot and a visible obstacle O moving on the same lane is depicted in Figure 3.2. The robot and the dynamic obstacle drive towards each other starting from their initial positions, denoted $Robot_1$ and $Obstacle_1$ respectively. Figure 3.2 shows that, initially, the obstacle is already inside the visual horizon of the robot's sensors. During the processing



Legend:

v_R - current robot velocity $Robot_i, i \in [1..3]$ - position of the robot on the lane $d^R_{Collision}$ - collision distance of the robot d^R_{Brake} - emergency braking distance of the robot in relation of the robot's current velocity c_R - safety net of the robot (dependent on the robot's reaction time) $d^R_{Collision}$ - collision distance of the robot h_R – visual horizon of the robot's sensors	v_O - current obstacle velocity $Obstacle_i, i \in [1..3]$ - position of the dynamic obstacle on the lane $d^O_{RobotBrake}$ - distance covered by the obstacle during the robot's braking time, while driving with the maximum velocity assumed for any dynamic obstacle in the environment $d^O_{RobotReaction}$ - distance covered by the obstacle during the robot's reaction time, while driving with the maximum velocity assumed for any dynamic obstacle in the environment d^O_{MaxVel} - maximum distance covered by the obstacle if it moves during the robot's reaction and braking time with the maximum velocity assumed for any dynamic obstacle in the environment

Figure 3.2.: Mobile Service Robot: Computation of the Collision Distance between the Robot and a Moving Obstacle.

of its sensor data, the robot travels the width of its safety net and reaches its second position $Robot_2$. At the same time, the dynamic obstacle arrives at position $Obstacle_2$, having traveled the distance $d^O_{RobotReaction}$. At this position, the obstacle is already situated inside the collision distance of the robot. Upon completion of the processing step, the robot triggers the brakes in order to avoid the collision and comes to a full stop in position $Robot_3$ on the lane. Meanwhile, the dynamic obstacle travels the distance $d^O_{RobotBrake}$ and reaches the position $Obstacle_3$. Note that the robot is stationary at the moment when the obstacle reaches the boundaries of its safety net, which is the desired behavior of the robot.

Observe that, according to Figure 3.2, the computation of the collision distance in Equation (3.3) accounts for both the maximum distance covered by the dynamic obstacle O during the robot's reaction time ($d^O_{RobotReaction}$) as well as the maximum distance traveled by the obstacle during the braking time of the robot ($d^O_{RobotBrake}$). As soon as the obstacle has passed by the robot and is outside of its safety net, the robot can resume the drive towards its destination.

Note that if the robot detects a stationary obstacle, the distance traveled by the obstacle becomes null and the respective collision distance is computed as follows:

$$d^R_{Collision} = d^R_{Brake} + c_R \tag{3.4}$$

The robot stops when it detects a stationary obstacle on the boundary of its safety net and checks whether another lane is safe to move to. If the robot finds another safe lane, then the robot changes to this lane and continues its drive towards its destination.

Each obstacle in the robot's environment belongs in one of the two categories at a given moment in time, i.e., it is either a *stationary obstacle* or a *dynamic obstacle*. Similarly to the mobile service robot, each obstacle O in its environment is characterized by its coordinates, (x_O, y_O), which encode the lane on which it is situated and its position on that lane, as well as its current velocity v_O. Also specific to each obstacle O is its maximum velocity, denoted v^O_{Max}, which represents a physical upper bound of the obstacle's current velocity. The stationary obstacles in the robot's environment are considered to be in a permanent state of rest, meaning that these obstacles do not move at all. As such, for any stationary obstacle $O_i, i \in \mathbb{N}_{>0}$, the following holds: $v_{O_i} = 0$ and $v_{O_i} = v^{O_i}_{Max}$. In turn, a dynamic obstacle $O_j, j \neq i, j \in \mathbb{N}_{>0}$ moves on its own lane, driving towards the robot from the opposite direction, with a current velocity $v_{O_j} \leq v^{O_j}_{max}$. In this simple example, each dynamic obstacle is considered to have as destination the origin of the Cartesian coordinate system, $(0,0)$, and once it has reached its destination, it will stop. On its way towards its destination, the dynamic obstacle can arbitrarily increase or decrease its velocity, but cannot change to another lane. When a dynamic obstacle stops, it becomes a stationary obstacle, entering a permanent state of rest. Since dynamic obstacles are not allowed to change their lanes, faster obstacles can overcome slower ones only by driving through them. In a similar manner, dynamic obstacles can pass by the robot by driving through it. However, at the moment when a dynamic obstacle drives through it, the robot must be stationary, i.e., $v_R = 0$, otherwise the event is categorized as an active collision caused by the robot (cf. Section 3.4).

3.2. Overall Development Process

The example system used for the problem analysis is a mobile service robot, commissioned to drive autonomously to a given destination. For safety-critical systems, such as autonomous robots, a plan-driven development process is appropriate (cf. [Som14a]). Thus, for the development of the example system the V-model [RB08] as displayed in Figure 3.3 has been followed. Notice that the V-model depicted in Figure 3.3 is a light version of the one used by Mauritz [Mau19].

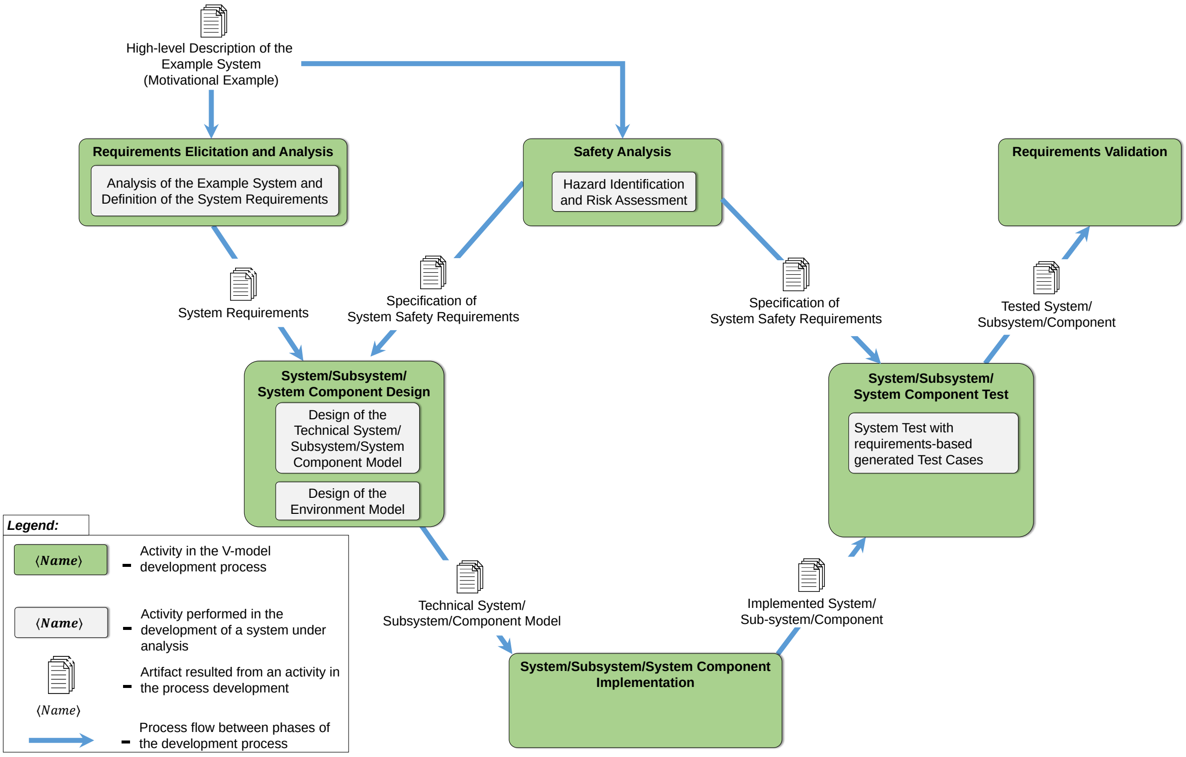

High-level Description of the Example System (Motivational Example)
Requirements Elicitation and Analysis
Analysis of the Example System and Definition of the System Requirements
Safety Analysis
Hazard Identification and Risk Assessment
Requirements Validation
System Requirements
Specification of System Safety Requirements
Specification of System Safety Requirements
Tested System/ Subsystem/Component
System/Subsystem/ System Component Design
Design of the Technical System/ Subsystem/System Component Model
Design of the Environment Model
System/Subsystem/ System Component Test
System Test with requirements-based generated Test Cases
Technical System/ Subsystem/Component Model
Implemented System/ Sub-system/Component
System/Subsystem/System Component Implementation
Legend:
⟨Name⟩ – Activity in the V-model development process
⟨Name⟩ – Activity performed in the development of a system under analysis
⟨Name⟩ – Artifact resulted from an activity in the process development
– Process flow between phases of the development process

Figure 3.3.: Overall System Development Process.

The first phase of the development process of the mobile service robot is *requirements elicitation and analysis*. This phase allows the requirements engineers to discover, understand, articulate, and document the users' needs and the constraints under which the system is supposed to operate (cf. Section 2.2). For the mobile service robot, the high-level description of the system's functionality presented in Section 3.1 serves as input to the requirements analysis, whose result is a set of system requirements gathered in a *functional system specification*.

The next phase of the development process, *system design*, is based on the requirements in the functional system specification and consists of three parts: functional system design, technical system design, and component specification (cf. Section 2.2). *Functional system design* allows the system requirements to be mapped to system functions and creates a *functional system architecture*, in which the interfaces and the communication protocols between the system functions as well as the interfaces between the system and its environment are defined. In *technical system design* it is decided which system functions are realized by hardware and which are implemented by software. The system functions realized in software are mapped to the corresponding software components, which constitute the *software system architecture*. Additionally, the *technical architecture* of the system is defined, which is comprised of the hardware components on which the system functions are executed together with the communication network between the hardware system components. During *component specification*, the interfaces with other components, the inner structure and component behavior are defined for each software component. For the mobile service robot, the functional system architecture consists of a technical system model and an environment model, which are connected with each other through the system-environment interface. The technical system model describes the desired behavior of the robot as specified in the functional system specification, while the environment model depicts the expected behavior of the obstacles in the robot's environment.

During *safety analysis*, the functional system specification and the functional system architecture are subject to a hazard analysis and risk assessment. The end result of the safety analysis is a *safety requirements specification*. The safety requirements specify measures which need to be applied in order to ensure that no hazard occurs, or if it occurs, to put in place specific mitigation techniques in order to minimize the harm severity and its consequences. The safety requirements are then fed back in the system design process and are incorporated in the functional system specification as additional requirements. For the mobile service robot as example system, a hazard analysis and risk assessment is performed on the basis of the high-level description of the example system in order to identify the safety requirements of the robot.

The next step in the system development is the *system implementation*. In this step, the functionality of the system and the safety mechanisms are primarily realized in software (cf. [Mau19]). The implementation of the software components is carried out under consideration of all the relevant aspects of the real world (cf. [SZ16]). This step takes as input the software system architecture modeled in a specific modeling language, and produces a full software implementation of the system. The implementation may be done manually or the source code may be generated automatically if the model of the software architecture is created with a toolchain which supports model-based development, e.g., MATLAB/SIMULINK [UPÇ12], ANSYS SCADE [CPP17]. For the mobile service robot, there is no implementation carried out. Instead, for the purpose of the problem analysis in this thesis, the technical system model and the environment model are run in parallel in a simulation, in order to illustrate and analyze the behavior of the robot in its environment.

After their implementation, each software component is tested with a series of test cases in what is called a *system component test*. Each test case covers a particular input/output combination, meaning that for a given set of test inputs a test case specifies an expected output. The goal of the system component test is to check that the component works correctly and completely as required by its specification (cf. [SLS14c]). In parallel to the software components, the respective hardware on which the software components are later deployed is also realized and tested (cf. [SZ16]). The software components which have successfully passed the component test, are gradually integrated into sub-systems and the sub-systems are in turn integrated with each other to build the whole software system. As the software integration progresses, *software integration tests* are performed in order to check whether the software components interact with each other as specified in the software architecture (cf. [SZ16]). The fully integrated software system is then installed on the target hardware platform and the whole system is checked for correct behavior in a comprehensive *system integration test* (cf. [SZ16]). Following this, the integrated system undergoes a *system test* in a physical test environment at the producer's site. While the system component test and the integration tests are performed against the respective technical specifications from the developer's point of view, the system test looks at the system from the customer's point of view and checks whether the system as a whole meets the specified system requirements (cf. [SLS14c]). Besides testing, other verification methods are adopted at times, e.g., model checking (cf. [BK08]). In the example of the mobile service robot, the parallel composition of the technical system model and the environment model is checked against the system requirements in the functional system specification with the help of a model checking tool.

The *requirements validation* is the last stage of testing in the system development process. In this step, the system stakeholders test the system on customer's site against the customer requirements (cf. [SLS14c]). The customer requirements define acceptance criteria which have been agreed upon in advance. The system must meet the predefined acceptance criteria before it is deployed and commercialized. In order to systematically derive test cases from the customer requirements, systematic methods such as automated test case generation via model checking can be applied (cf. [AHDR18]). For the mobile service robot, no requirements validation has been performed. Instead, the parameters of the environment model are manipulated in order to emulate unforeseen situations, which may appear during testing in a physical environment on customer's site, and the technical system model is checked together with the new environment model against the specified requirements.

3.3. Requirements Elicitation and Analysis

Requirements elicitation and analysis is the first step in the development phase and is performed using as input the high-level description of the system's functionality, introduced in Section 3.1. An informal specification of the system requirements is derived and presented in Section 3.3.1, after which the requirements are formalised in Section 3.3.2.

3.3.1. Informal Specification of System Requirements

The system in this example is a robot commissioned to drive towards and reach a given destination, without actively colliding with any obstacle in its environment. There are two decisions that were made in the high-level description of the mobile robot with regard to the representation of the environment. These decisions have an impact on the tasks of system design thereafter. Firstly, remember that the environment is defined as a subset of the two-dimensional space built by the Cartesian coordinate system. Secondly, the majority of the physical characteristics of both the robot and the obstacles as physical objects are abstracted from and they are considered to be discrete points in the Cartesian two-dimensional space. However, the robot and the obstacles in its environment are considered to retain one physical trait, i.e., each is considered to have a current positive velocity limited by a specific maximum value. Furthermore, it is considered that the robot's environment is populated with dynamic and stationary obstacles. Note that in case of stationary obstacles, both the current and the maximum obstacle velocity are null.

The robot's sensors are not considered to be flawless. Moreover, it is considered that the sensors' field of view is limited by its specific maximum range, denoted by h_R, which is set in the sensors' configuration. Note that the maximum range of the sensors is the maximum distance to which the sensor can pick up reflective light and return an accurate distance measurement.

The specification of system requirements and safety requirements is often done and organized through informal textual documents (cf. [AHDR18]). In the best case, the informal textual documents are broken down into lists of individual requirements and are maintained using a dedicated tool, e.g., IBM DOORS (cf. [AHDR18]). Whether the requirements are maintained as simple textual documents or with the help of dedicated tools, the requirements are initially written in natural language, which is by definition imprecise and ambiguous. Ambiguity in the requirements can cause various system stakeholders to have different understandings of the required system functionality (cf. [RS14]). One method to avoid ambiguities in the requirements' formulation is to use requirements templates or requirements patterns. A requirements template is basically a construction plan which lays down the building blocks necessary for the formulation of requirements (cf. [JPQ$^+$16], [RS14]). In [JPQ$^+$16], the basic rules for writing requirements are presented: (1) requirements are always described in the active form, (2) requirements are always written as complete sentences, (3) requirements express processes or activities with the help of process verbs, e.g., drive or brake, and (4) exactly one requirement is formulated for each process verb. These rules have been observed and applied in the formulation of the system requirements for the mobile service robot.

In the functional system specification of the robot, a process verb, e.g., drive, accompanies a modal verb, e.g., shall. The modal verb shows the different legal meanings that a requirement may have for various stakeholders. In [JPQ$^+$16], three modal verbs are considered for the formulation of requirements: *shall*, *should*, and *will*. All requirements formulated with *shall* are compulsory for the system implementation. Requirements formulated with *should* represent a stakeholder's wish, are not binding and do not have to be implemented. However, their implementation increases stakeholder satisfaction, and their documentation improves communication between the development team and the system stakeholders (cf. [JPQ$^+$16]). A requirement formulated with *will* serves as preparation for a functionality which is planned to be integrated in the future. The development team is obliged to consider this requirement in the system implementation, even if the implementation of this functionality is not tested at first (cf. [JPQ$^+$16]). All requirements of the mobile service robot are formulated with the modal verb *shall*, and therefore are compulsory in the system implementation.

To begin with, several high-level functional requirements, depicted in Table 3.1, define which actions the mobile service robot is allowed to execute and which actions are excluded from the functionality of the robot. Thus, in order to reach its destination the robot must be able to drive forwards (FR1), but also change to another lane in case it needs to overcome an obstacle or avoid a collision danger (FR2). In this respect, dynamic obstacles in the robot's environment are more restricted because they cannot change lanes (FR2). In order to simplify the dynamics of the example system, the mobile service robot cannot drive backwards, a constraint which applies also to the dynamic obstacles in the robot's environment (FR3). Remember that the mobile service robot and the obstacles in its environment are stripped of their physical dimensions and are abstracted to discrete points in the two-dimensional Cartesian space. This makes it possible to talk about faster obstacles overcoming slower obstacles by driving through them (FR4). However, this is does not apply to the mobile service robot, because it would represent a safety risk for the robot. Furthermore, there is another aspect in which the mobile service robot is distinct from the dynamic obstacles in its environment. While the mobile service robot, once stopped, may resume its driving, dynamic obstacles remain stationary once they have stopped moving (FR5). The field of view of the robot's sensors stretches up to a specific distance in front of the robot, which is configurable in advance. Due to the physical limitations of its sensors, there are areas of the environment which the robot cannot perceive. These areas of the environment are called the blind spots of the robot. Figure 3.1 gives a visual representation of the field of view of the robot's sensors, i.e., the gray area spanned up to the finite horizon h_R, and implicitly, of the blind spots of the robot, i.e., any area situated outside sensor field of view of the robot. Any obstacle situated in a blind spot of the robot cannot drive in the same direction as the robot (FR6). Considering the physical overview of the motivational example given in Figure 3.1, this means that any obstacle situated in these blind spots is not allowed to follow the robot and overtake the robot from behind or from the neighboring lane. Notice that this prohibits also dynamic obstacles to lead the robot, since a leading obstacle becomes a following obstacle if the robot chooses to overtake it. Finally, the robot cannot jump over several lanes, in the same way that dynamic obstacles in the robot's environment are not permitted to execute any lane jumps (FR7).

On a functional level, the mobile service robot has two main operation modes which are derived from the high-level description of the system's functionality in Section 3.1: normal operation and collision avoidance.

The *normal operation mode* of the mobile service robot consists of all the activities which the robot performs as a system in order to achieve its goal. The system requirements regarding the normal operation of the robot are depicted in Table **??**. In the motivational example, the goal of the mobile service robot is to drive autonomously towards a given destination (FR8) and reach it (FR9). To drive towards its destination, the robot must be able to accelerate up to its specific maximum speed (FR8.1) and maintain this speed as long as no collision danger is detected (FR8.2). Should the robot be forced to stop due to collision danger, it must be able to resume its driving if the obstacle has passed

Table 3.1.: Mobile Service Robot: Requirements for Allowed and Forbidden System Actions in the System's Environment.

ID	Requirement Text
FR1	The robot shall be able to drive forwards, in an environment where dynamic obstacles also move forwards.
FR2	The robot shall be able to change to a safe target lane, in an environment in which dynamic obstacles do not change lanes.
FR3	The robot shall not be able to drive backwards, in an environment in which dynamic obstacles do not drive backwards.
FR4	The robot shall be able to overcome obstacles only by changing to another lane, in an environment in which dynamic obstacles pass by other obstacles by driving through them.
FR5	If stopped, the robot shall be able to resume its driving, in an environment in which dynamic obstacles which have become stationary do not resume their movement.
FR6	The robot shall be able to perceive the space in front of itself up to a specific configurable sensor horizon limit, in an environment in which obstacles situated in the blind spots of the robot do not drive in the same direction as the robot.
FR7	The robot shall not be able to jump over lanes, in an environment in which obstacles do not jump over lanes.

by (FR8.3) or if the robot detects a safe target lane on which it can continue its driving (FR8.4). Furthermore, if the robot detects a safe target lane, than it must be able to change to this lane in order to continue the drive towards its destination (FR8.5). As the robot comes nearer its destination it must be able to slow down (FR9.1), and to stop, when it has reached its destination (FR9.2).

The *collision avoidance mode* of the mobile service robot contains all the activities which the robot must carry out in order to avoid collision danger. Not only does a collision of the mobile service robot with an obstacle in its environment represent a safety risk and can cause damages both to the robot as well as to the obstacle, it also impedes the robot from carrying out its task and reaching its predefined destination. For this reason collision avoidance belongs to the functionality of the mobile service robot.

The requirements which address collision avoidance in the functional system specification are depicted in Table **??**. In order to avoid collision danger, the mobile service robot must be able to first detect the collision danger in front of itself inside its sensors' horizon (FR10), and then to apply collision avoidance measures (FR12).

Table 3.2.: Mobile Service Robot: Requirements for the System's Normal Operation Mode.

ID	Requirement Text
FR8	The robot shall drive towards a given destination.
FR8.1	The robot shall accelerate until it reaches a specific maximum speed, as long as it has not reached its destination and as long as it has not detected any collision danger.
FR8.2	The robot shall continue driving with its specific maximum speed, as long as it has not reached its destination and as long as it has not detected any collision danger.
FR8.3	The robot shall resume driving, if it has stopped due to collision danger and if it has not reached its destination and if it the obstacle has passed by.
FR8.4	The robot shall resume driving, if it has stopped due to collision danger and if it has not reached its destination and if it detects at least one safe target lane.
FR8.5	The robot shall resume accelerating, if it detects at least one safe target lane during its brake maneuver.
FR9	The robot shall reach the given destination.
FR9.1	The robot shall start braking, when it approaches its destination.
FR9.2	The robot shall stop, when it reaches its destination.

There are two ways in which the mobile service robot can avoid collision: (1) change to a safe target lane, in case such a lane is detected (FR12.1), or (2) start to brake, in case no safe target lane is detected (FR12.2). For it to be able to apply these measures, the robot must also be able to detect safe target lanes (FR11). Even though the robot may be forced to brake in order to avoid collision danger, it does not necessarily apply immediately the emergency brakes which bring it to a full stop. Instead, the robot must be able to check whether further drive with reduced velocity is possible (FR13).

If further drive is possible, the robot must be able to maintain the new speed (FR13.2), in order to cover as much ground as possible in its drive towards its destination. Otherwise, the robot must apply the brakes until it comes to a full stop (FR13.3). Further drive is considered possible if no collision danger is detected on the ego lane (FR13.1). Once the robot has come to a standstill due to collision danger, it shall remain at rest as long as it has not detected a safe target lane and the obstacle has not passed by (FR14).

Table 3.3.: Mobile Service Robot: Requirements for the System's Collision Avoidance Mode.

ID	Requirement Text
FR10	The robot shall detect collision danger in front of itself in its sensor horizon.
FR11	The robot shall detect safe target lanes.
FR12	The robot shall apply collision avoidance measures, if it detects collision danger on the ego lane.
FR12.1	The robot shall change to a safe target lane, if it detects collision danger on the ego lane and if it detects a safe target lane.
FR12.2	The robot shall brake, if it detects collision danger on the ego lane and if it does not detect any safe target lane.
FR13	The robot shall check if further drive with reduced speed is possible.
FR13.1	The robot shall consider that further drive with reduced speed is possible if it detects no collision danger on the ego lane.
FR13.2	The robot shall reduce its speed and maintain it, if further drive with reduced speed is possible.
FR13.3	The robot shall brake until it stops, if further drive with reduced speed is not possible.
FR14	The robot shall remains at rest, if it has stopped due to collision danger and as long as it has not detected a safe target lane and as long as the obstacle has not passed by.

3.3.2. Formal Specification of System Requirements

System requirements in the functional system specification are defined as conditions or capabilities which a system must meet or possess in order for the end users to solve a problem or achieve an objective (cf. [BHKS12]). At the same time, it is important for a system to meet its requirements, in order to achieve compliance with a contract, regulation, standard or any other form of formally imposed document (cf. [ISO10]). The system requirements constitute the basis for capturing and communicating the needs of different stakeholders, managing the scope and the system boundaries, and verifying and validating the system (cf. [BHKS12]). Natural language is predominant in requirements documents, as it is considered to be the most effective in gaining the customers understanding and agreement (cf. [BHKS12]). Nevertheless, systematic analyses are required for the purpose of verification and validation of the system. Usually, such analyses are aided by formal methods, which encompass the formalization of system requirements in a specific formal language, e.g., temporal logic. This raises, however, additional issues. One one hand, customers are rarely prepared to sign a contract in which the requirements specification is written only in a formal language (cf. [BHKS12]). On the other hand, the system development team needs a precise, unambiguous specification of the system's functionality, which can be used as reference during the verification and validation process. Furthermore, requirements formalization is not a trivial task (cf. [Hol02]). It requires expertise in the area of formal methods as well as knowledge of the application domain in which the system requirements are formulated (cf. [Buz19]).

Several research works have been concerned with bridging the gap between natural language requirements and formal language specification. One area of research in this direction is goal-oriented requirements engineering, which "uses goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting and modifying requirements"[Lam01]. Goals are defined as objectives which the system under development is required to achieve. The main activity in goal-oriented requirements engineering is requirements elaboration through the refinement and operationalization of goals. Darimont et al. [DL96] introduce a technique which provides support for formal reasoning about goals and an approach for their refinement and operationalization. This approach uses generic refinement patterns organized in a library which also defines strengthening and weakening relationships between abstract goal formulations. Goals identified during the requirements engineering phase are refined through an AND/OR direct acyclic graph (cf. [DL96], [PMM+07]). Parent goals, defined at a higher level, are refined into child goals by asking *How*-questions, while parent goals can be identified by asking *Why*-questions. Goals defined at a higher level are more coarse-grained and require usually several agents - humans, software programs, or devices - for their realization, while low level goals are situated on technical level and can be realized by individual agents. Requirements are terminal goals which cannot be refined any further and which fall into the responsibility of a single agent belonging to the system under development. Various temporal patterns expressed in temporal logic, e.g.,*achieve* or *maintain*, are used in order to obtain the goal refinement structure (cf. [DL96], [Lam01]). These patterns are extended in [PMM+07] with real-time temporal constructs. At the end of the goal refinement, a goal model is obtained, which prescribes a set of intended system behaviors, where a behavior is a temporal sequence of system states (cf. [PMM+07]). Goals are operationalized as a set of operations or actions. The applications of these operations are state transitions along the behaviors prescribed by the goal model. The operationalization of goals defines for each state transition a triple consisting of precondition, trigger condition and postcondition (cf. [PMM+07]).

Other research works focus specifically on the problem of requirements formalization in temporal logics. Buzhinsky [Buz19] carries out a survey of approaches for the formalization of requirements into temporal logics for the purpose of model checking. The formalization of requirements poses several challenges even for experts in the field of formal methods, e.g., ambiguity of natural language and the changes occurred in the grammatical structure of natural language requirements when these are translated in a temporal logic (cf.

[Buz19]). The surveyed approaches for requirements formalization fall into one of the two categories: (1) direct formalization of natural language requirements and (2) requirements formalization through statistical machine translation.

Direct formalization of requirements comprises several methods, each with its own approach angle towards the problem of requirements formalization. For instance, pattern-based approaches use the concept of *property specification pattern*, inspired by the patterns in software design [Gam11], in order to establish a direct connection between requirements formulated in natural language and certain types of temporal logic formulas (cf. [Buz19]). A property specification pattern is defined as a "generalized description of a commonly occurring requirements with respect to the permissible state or event sequences in a finite-state model of a system" [DAC98, DAC99]. Dwyer et al. [DAC98] propose a system of property specification patterns, which is organized as a hierarchy based on the semantics of the specific patterns. Three categories of patterns are defined in this pattern system: (1) occurrence, e.g., the absence pattern, (2) ordering, e.g., the precedence pattern, and (3) compound, e.g., chain precedence which is a generalization of the precedence pattern (cf. [DAC98]).

Flake et al. [FMR00] and Konrad and Cheng [KC05] use structured English grammars for selected subsets of the English language in the context of real-time systems to translate requirements written in natural language into specific patterns based on the system of patterns proposed by Dwyer et al. [DAC98]. Structured English grammars can also be applied for the specification of complex robotic missions (cf. [MTP$^+$19]). The structured English grammar is used in conjunction with patterns as basic building blocks and operators which allow the composition of patterns (cf. [MTP$^+$19]). The patterns are gathered in a catalog of patterns and are classified according to three concerns: (1) core movement patterns, which express how robots should move in an environment, (2) avoidance patterns, by which the robot's movements are constrained in order to avoid occurrence of undesired behavior, and (3) trigger patterns, which describe reactive behavior based on stimuli (cf. [MTBP19], [MTP$^+$19]). Bitsch proposes a catalog of safety patterns used for the formal specification of safety requirements of industrial automation systems (cf. [Bit01]). For each safety pattern, Bitsch gives the natural language specification accompanied by the formal specification in CTL. In the classification of the safety patterns, Bitsch takes into consideration the terminology used in industrial automation systems, the constructs specific to CTL as a formal specification language, and formulation of the safety requirements in natural language (cf. [Bit00]). Using this criteria, the author differentiates safety requirements in several categories: static vs. dynamic requirements, general guarantee of occurrence vs requirements on execution sequences, requirements on constraint behavior, on permitted behavior and on conditional guarantee for constraint behavior, requirements on beginning and duration of validity and of exact chronological succession, real-time requirements for time triggered and event triggered system (cf. [Bit00]).

Other direct formalization approaches use formal grammars as fixed translation models between natural language requirements and their formal specification. Remaining in the domain of task and motion planing for robotic systems, a structured subset of English serves as an intermediate layer between the logical formalism and the natural language in order to formulate the user specification for task and motion planing of robot controllers (cf. [KGFP08]). Kress-Gazit designs the structured, controlled language using a formal grammar, and uses it in conjunction with LTL to generate robot actions and trajectories which satisfy the user specifications expressed in natural language (cf. [KGFP08]). To translate requirements from natural language to temporal logic, Fantechi et al. [FGR$^+$94] build a context-free grammar and a dictionary based on a corpus of natural language requirements from the domain of reactive systems. The translation process takes place in two steps: sentence analysis to extract relevant information from the input sentence, e.g., time quantification, and construction of a parsing tree for the input sentence, from which the temporal logic formula is then generated. Nelken and Francez [NF96] use discourse representation theory as an intermediate level between natural language and temporal logic. The proposed method takes specification discourses as input and it parses them with a parser written on the basis of a unification-based grammar formalism. The result of the parser, a discourse representation structure, is used as basis for the generation of the temporal logic formula (cf. [NF96]).

Nikora uses machine learning and natural language processing techniques to differentiate between temporal and non-temporal requirements (cf. [Nik05]). Furthermore, Nikora and Balcom take Dwyer's patterns system to categorize the temporal requirements under analysis and use learning models built with machine learning techniques to identify the most frequently occurring LTL patterns within a set of temporal specifications. In total, a number of eight requirements patterns are identified, e.g., the existence, universality, and precedence patterns, similar to those in Dwyer's system of property specification patterns (cf. [NB09]).

Mauritz proposes a pattern-based approach for the structuring of the system requirements (cf. [Mau19]). The requirements are formulated in an informal specification in a controlled natural language, with one sentence per system requirement. In his approach, Mauritz superimposes his requirements pattern over the system requirements, which allows the identification of various parts of the domain knowledge, e.g., subject of the system requirement, its properties, system actions and relationships with other objects. The parts of the domain knowledge are then organized in types inside a domain model, which is then used as input for the formalization with typed first-order logic (cf. [Mau19]).

Another approach which aims to bridge the gap between natural language requirements and their formal specification is represented by Catia Stimulus. The Stimulus language enables the specification of requirements in a manner which resembles textual requirements specification. For this purpose, the language uses the notion of *system* to encapsulate statements and macros, which accept statements as parameters, in order to define scoping and typing rules (cf. [JG16]).

In a similar approach to that of Mauritz [Mau19] and that of Jeannet and Gaucher [JG16], this thesis defines domain specific concepts for the purpose of requirements formalization for the mobile service robot. These concepts are translated into first-order logic predicates, since they are later used in further computations throughout this thesis. The first concepts to be defined are concerned with the robot approaching its destination and the robot reaching its destination. Notice that the destination of the mobile service robot is specified by Cartesian coordinates of its goal position.

Definition 3.3.1 - Robot Near Destination
Let (x_R, y_R) be the coordinates of the robot's current position and (x^R_{Goal}, y^R_{Goal}) be the coordinates of the robot's given destination. The robot is considered to be *near* its *destination* if $x_R + d^R_{Brake} = x^R_{Goal}$, where d^R_{Brake} is the braking distance of the mobile service robot with respect to the robot's current speed v_R. ■

Definition 3.3.2 - Robot Reached Destination
Let (x_R, y_R) be the coordinates of the robot's current position and (x_G, y_G) be the coordinates of the robot's goal position. The robot is considered to have *reached* its *destination* if $x_R = x^R_{Goal}$. ■

The first-order logic predicates corresponding to Definitions 3.3.1 and 3.3.2 are defined in Equations (3.5) and (3.6) respectively.

$$\mathit{nearDestination}: \quad (x_R + d^R_{Brake} == x^R_{Goal}) \tag{3.5}$$

$$\mathit{destinationReached}: \quad (x_R == x^R_{Goal}) \tag{3.6}$$

Notice that the predicate in (3.6) does not require the robot's lane to coincide with the destination lane at the end of the robot's journey. This can be attributed to the decision process used by the robot when it chooses a lane change over braking to a standstill in case of collision danger and several lanes are safe.

Since it aims to cover as much ground as possible, the robot changes to the lane that offers the longest distance to the visible obstacle situated on that lane. The consequence of this is that there are executions of the overall system model in which, at the end of its journey, the robot does not land on the destination lane, i.e., $y_R \neq y^R_{Goal}$.

The next concept which needs to be defined is that of collision danger. Notice that the robot can perceive an obstacle as a potential collision danger only if the obstacle is visible to the its sensors. Therefore, it is necessary to define beforehand the notion of visible obstacle, which is done in Definition 3.3.3.

Definition 3.3.3 - Visible Obstacle
Let $n \in \mathbb{N}_{>0}$ be the number of obstacles in the environment of an autonomous mobile robot. An obstacle $O_i, i \in [1..n]$ is considered to be a *visible obstacle* for the robot if and only if the following two statements are true simultaneously:

1. $-c_R \leq x_{O_i} - x_R \leq h_R$,
2. $\forall O_j, O_j \neq O_i, j \in [1..n]$:
   a) if $-c_R \leq x_{O_i} - x_R < 0$ and $-c_R \leq x_{O_j} - x_R < 0$ then $x_{O_i} - x_R > x_{O_j} - x_R$.
   b) if $0 \leq x_{O_i} - x_R \leq h_R$ and $0 \leq x_{O_j} - x_R \leq h_R$ then $x_{O_i} - x_R < x_{O_j} - x_R$.

■

Notice that the first statement of Definition 3.3.3 requires that the visible obstacle is situated inside the robot's sensor horizon, or if the obstacle has already passed by the robot, inside its safety net.

The second statement requests that the visible obstacle is in the direct line of sight of the robot's sensors, i.e., no other obstacle is interposed between itself and the robot. As an example, consider the physical overview of the robot's environment in Figure 3.1. Once the dynamic obstacle O_2 enters the sensor horizon of the robot it becomes a visible obstacle for the robot. However, the same cannot be said about the obstacle O_3. The obstacle O_2 blocks the line of sight of the robot's sensors, which leads to the obstacle O_3 not being a visible obstacle for the robot even when it is situated inside the visual horizon of its sensors.

The first-order logic predicate corresponding to Definition 3.3.3 is introduced in Equation (3.7):

$$\begin{aligned} \mathit{VisibleObstacle} : \forall O_j . O_j \neq O_i \wedge \{&[\neg\, (-c_R \leq x_{O_i} - x_R \ \wedge\ x_{O_i} - x_R < 0 \\ &\wedge\ -c_R \leq x_{O_j} - x_R \ \wedge\ x_{O_j} - x_R < 0) \\ &\vee\ x_{O_i} - x_R \ > \ x_{O_j} - x_R] \vee \\ &[\neg\, (0 \leq x_{O_i} - x_R \ \wedge\ x_{O_i} - x_R \leq h_R \\ &\wedge\ 0 \leq x_{O_j} - x_R \ \wedge\ x_{O_j} - x_R \leq h_R) \\ &\vee\ x_{O_i} - x_R \ < \ x_{O_j} - x_R]\} \end{aligned} \tag{3.7}$$

in which $O_i, i \in [1, n]$ is the visible obstacle and $O_j \neq O_i, j \in [1, n]$ is a random obstacle in the robot's environment.

Since the dynamic obstacles in the robot's environment cannot change their lane, the obstacles situated on the robot's lane are the only ones which represent potential collision dangers for the robot. Thus, it becomes important to distinguish the obstacles on the ego lane from all the other obstacles in the environment. For this purpose, the parameterized first-order logic predicate defined in Equation (3.8) is used:

$$\mathit{VisibleObstacleOnLane}(y) : \mathit{VisibleObstacle} \ \wedge\ (y_{O_i} \ == \ y) \tag{3.8}$$

Finding out whether an obstacle is visible on the ego lane is done with a call of the predicate *VisibleObstacleOnLane*(y_R) with the ego lane as parameter y_R, in which every appearance of the variable y is substituted with the variable y_R.

Having defined the notion of visible obstacle situated on ego lane, it is now possible to establish whether an obstacle constitutes a collision danger for the mobile service robot. This is done through Definition 3.3.4.

Definition 3.3.4 - Collision Danger

Let $n \in \mathbb{N}_{>0}$ be the number of obstacles in the environment of an autonomous mobile robot. An obstacle $O_i, i \in [1..n]$ is considered to be a *collision danger on a given lane* for the robot if and only if the following two statements are true simultaneously:

1. the obstacle O_i is a visible obstacle on the given lane, and
2. the obstacle O_i is situated inside the collision distance of the robot.

■

The corresponding first-order logic predicate is depicted in Equation (3.9):

$$\begin{aligned} \mathit{CollisionDangerOnLane}(y) : &\mathit{VisibleObstacleOnLane}(y) \ \wedge \\ &(0 \leq x_O - x_R) \ \wedge\ (x_O - x_R \leq d^R_{\mathit{Collision}}) \end{aligned} \tag{3.9}$$

Notice that, since dynamic obstacles are not allowed to change lanes (FR2), only those obstacles situated on the ego lane represent a potential collision danger for the mobile service robot. However, the first-order logic predicate defined in Equation (3.9) allows more flexibility and can be used for checking any lane in the robot's environment for collision danger due to its parameter y. Checking whether there is a potential collision danger on the ego lane is done by calling the predicate *CollisionDangerOnLane* with the ego lane y_R as parameter, which substitutes every occurrence of the variable y in the predicate *CollisionDangerOnLane* with the variable y_R.

The mobile service robot has two maneuvers at its disposal which it can use in order to avoid collision danger. The first maneuver enables the robot to brake until it comes to a full stop, while the second one lets the robot change to a safe target lane. The concept of safe target lane for an autonomous mobile robot is introduced in Definition 3.3.5.

Definition 3.3.5 - Safe Target Lane
Let $n \in \mathbb{N}_{>0}$ be the number of obstacles in the environment of an autonomous mobile robot. A lane in the environment of the robot is considered to be a *safe target lane* if and only if there is no visible obstacle on the lane which is situated inside the collision distance of the robot. □

In order to facilitate the use of Definition 3.3.5 throughout this work, a first-order logic predicate corresponding to the notion of safe target lane must be introduced. Notice that for the definition of this predicate, an auxiliary predicate must be defined, which denotes the visible obstacles situated outside the robot's collision distance, as shown in Equation (3.10).

$$\begin{aligned} \mathit{VisibleObstacleOutsideCollisionDistance} : \mathit{VisibleObstacle} \wedge \\ (x_O - x_R > d^R_{\mathit{Collision}}) \end{aligned} \tag{3.10}$$

Notice that there are two situations in which a lane is considered to be safe according to Definition 3.3.5. Firstly, in the trivial case, a target lane is considered to be safe if there are no visible obstacles situated on this lane. Secondly, in the more complex case, if there are visible obstacles which occupy the target lane, they must be situated in front of the robot outside of the area spanned by the robot's collision distance, or behind the robot and outside of its safety net.

The two auxiliary predicates introduced in Equation (3.8) and Equation (3.10) are used in the construction of the safe target lane predicate, as shown in Equation (3.11):

$$\begin{aligned} \mathit{SafeTargetLane}(y) :\neg\, & \mathit{VisibleObstacleOnLane}(y) \vee \\ & (\mathit{VisibleObstacleOnLane}(y) \wedge \\ & \mathit{VisibleObstacleOutsideCollisionDistance}) \end{aligned} \tag{3.11}$$

Liveness Properties

The system requirement FR9 states that the robot shall reach a given destination. Such a requirement is formalized as a liveness property. A *liveness property* asserts that "something good eventually happens", i.e., a specific condition is guaranteed to hold or an event is guaranteed to happen at some future moment in time. See Chapter 2 for an introduction on liveness properties in the context of design-time verification. Informally, the liveness property corresponding to requirement FR9 can be formulated as follows:

Liveness Property (Informal Specification). The robot shall eventually reach its destination.

CTL is chosen for the specification of this liveness property. In CTL [CE81], liveness can be expressed with the path formula $AF\psi$. The operator A is the universal path quantifier, which requires that a certain property holds on all computation paths of a system model. Thus, the property $AF\psi$ denotes that on all computation paths $F\psi$ holds. That is, it states that on all computation paths there is a state which satisfies ψ that is eventually reached. In the case of the robot's liveness property, ψ is the predicate *destinationReached*, as illustrated in Equation (3.12).

$$\theta : AF\ (\mathit{destinationReached}) \tag{3.12}$$

Notice that the liveness property is very similar to the existence pattern identified by Dwyer et al. [DAC98], which states that a given state or event must occur within a scope. In the example of the mobile service robot, the scope is represented by all computation paths of the robot's system model.

The property in Equation (3.15) expresses that the robot inevitably reaches its destination at some time in the future. Notice that, for the liveness property to be satisfied, it is considered sufficient that the robot reaches the position of its destination point, and not necessarily the lane on which this point is situated. Remember that this can be attributed to the decision process of the robot and its goal to cover as much ground as possible on its drive towards its destination. When the robot has to perform a lane change and can choose from several safe target lanes, it chooses the lane which offers the longest distance without visible obstacles in sight. Thus, there exist computation paths on which the robot reaches the x-position of its destination, but not the lane, that is the y-position, on which the destination is situated.

3.4. Safety Analysis

The mobile service robot is a safety-critical system which takes mission- and safety-critical decisions while driving through a heterogeneous dynamic environment towards a given destination. Such environments are characterized by the presence of a wide range of obstacle types, including humans. The decisions of the mobile service robot have the potential to threaten the safety of the robot itself and potentially cause harm to the persons and other obstacles in the robot's environment. Harm is defined as physical injury or damage to the health of persons by ISO 26262 [Int11b] and ISO/PAS 21448 [Int19], while ANSI D.16.1 [Ass17] associates two categories to the notion of harm, namely injury brought on persons and damage caused to property. Koopman and Wagner [KW16] point out to a safety principle established in the safety-critical application domains, which states that safety must be demonstrated and not assumed (cf. [KW16]). This means that safety-critical systems should be considered unsafe, unless convincing arguments are made for the safety of the system under analysis (cf. [KW16]). National and international authorities impose regulations for autonomous safety-critical systems which require the system manufacturers to provide sufficient proof that their systems are safe before receiving permission for commercialization.

System safety describes the extent to which the system under analysis is free of hazards (cf. [GK19]), which further translates into the absence of unreasonable risks of physical harm (cf. [Int11b]) for the system as well as for the environment in which it operates. In order to bring up sufficient proof for the system safety, a process of safety analysis must be carried out, which uses specific methods for the identification, analysis and classification of hazards and assessment of the subsequent risks for the system. The international standards ISO 26262 [Int11g] and ISO/PAS 21448 [Int19] recommend several methods for the analysis of hazards and assessment of risks, e.g., HAZOP, FTA, or FMEA (cf. Section 2.2). Notice that hazards may occur due to factors related to the system itself, e.g., malfunctioning behavior or deficiencies in the specified behavior (cf. [Int11b]), but also due to factors external to the system, e.g., “passive” infrastructure or environmental conditions in which the system operates (cf. [Int19]). Mauritz performs a safety analysis for lane change assistant system, using the recommendations and the process defined by ISO 26262 (cf. [Mau19]). The prerequisites for the safety analysis required by ISO 26262 is the definition of the system under analysis and its preliminary architectural information (cf. [Int11c]). Through FTA, fault events are tracked to specific system components defined in the hardware and software architecture of the lane change assistant, e.g., sensors, perception functions, situation assessment and behavior planing, and actuators (cf. [Mau19]).

For the example of the mobile service robot, the high-level description of the system functionality given in Section 3.1 and the system requirements described in the functional system specification in Section 3.3 are taken as input for the HARA method. The HARA analysis performed on the mobile service robot is shown in Table 3.4 and defines a classification of the aspects which are relevant for the hazard analysis together with equivalence classes or parameter values. The most important aspects for the hazard analysis are:

- the *location* where the mobile robot operates, e.g.,

$$\mathit{Location} := \{\mathit{Indoors}, \mathit{Outdoors}\}$$

- the *geometry* of the physical world in which the robot moves specified through the parameters the number of lanes n_{Lanes} and the ground inclination α_{Ground}, e.g.,

$$\mathit{PhysicalWorldGeometry} := \{(n_{Lanes} \geq 3,\ \alpha_{Ground} = 0°), \dots\}$$

- the *driving conditions*, e.g.,

$$\mathit{DrivingConditions} := \{(\mathit{Dry}, \mathit{Non\text{-}slippery}), (\mathit{Wet}, \mathit{Slippery}), \dots\}$$

- the *environment* in the sensor horizon of the robot, e.g.,

$$\begin{aligned} \mathit{Environment} := \{&(\mathit{StraightRoad}, \mathit{DynamicObst}),\\ &(\mathit{StraightRoad}, \mathit{StationaryObst}), \dots\} \end{aligned}$$

- the *system usage*, e.g., driving towards a predefined destination,
- the *system behavior*:

$$\begin{aligned} &\mathit{SystemBehavior} = \mathit{RobotLaneChangingBehavior} \times \mathit{RobotDrivingBehavior}\\ &\mathit{RobotDrvingBehavior} = \{\mathit{Accelerating}, \mathit{FullSpeedDrv}, \mathit{ReducedSpeedDrv},\\ &\mathit{Braking}, \mathit{Stopped}\}\\ &\mathit{RobotLaneChangingBehavior} = \{\mathit{LeftLaneChange}, \mathit{RightLaneChange},\\ &\mathit{LaneKeep}\} \end{aligned}$$

- the *environment behavior*, which refers to the behavior of the obstacles situated in the robot's sensor horizon, e.g., a dynamic obstacle overtaking a slower obstacle.

The system behavior consists of a combination of the lane changing behavior and the driving behavior of the mobile service robot. This combination is modeled as the Cartesian product between the set of actions or related to the robot's lane changing behavior and the set of actions describing the robot's driving behavior.

The lane changing behavior of the robot is triggered in case a collision danger is detected. Remember that in case of collision danger the robot, the robot has two alternatives: either change to a safe target lane or brake to a standstill and wait until the dynamic obstacle which triggers the collision danger has passed by. Notice that in the search for a safe target lane, the lane on the right (*RightLaneChange*) or the lane on the left of the robot (*LeftLaneChange*) may turn out to be safe for the robot to continue its drive. If however no other safe lane is detected, then the robot keeps its current lane (*LaneKeep*). This is consistent with the requirement FR7 in Table 3.1, which does not allow the robot to jump over lanes.

The driving behavior of the robot encompasses actions which allow the robot to accelerate (*Accelerating*), drive with maximum speed (*FullSpeedDrv*), brake in case of collision danger or when it approaches its destination (*Braking*), drive with reduced velocity rather than brake to a standstill (*ReducedSpeedDrv*) or stop (*Stopped*).

Notice that the selection of the parameters relevant for the hazard analysis is dependent on the task which the system is supposed to perform and on the environment in which the system operates. In fact, the presence of certain aspects in the HARA analysis may warrant the consideration of other aspects which are, in a way, dependent on the former. Take for example the location and driving conditions. It makes sense to consider the driving conditions when the operational environment is located outdoors. However, the mobile service robot is considered to operate in an indoor environment. As such, the driving conditions do not play an important role and are not further considered in HARA analysis carried out in Table 3.4. The values of the relevant parameters are combined with each other in order to create individual situations, with one situation being depicted in one table row of Table 3.4. There are an uncountable number of ways in which the obstacles in the robot's environment may behave. Thus, neither the list of identified situations nor the list of selected parameters are considered to be complete.

For every identified situation, the HARA analysis defines a possible hazard and the potential effect this hazard may have. Take for the example situation H1 depicted in Table 3.4, in which the mobile service robot drives indoors on a straight road with dynamic obstacles and with two or more lanes. Imagine there is a dynamic obstacle O_1 moving with velocity v_{O_1} on the ego lane towards the robot. Furthermore, behind obstacle O_1 and on the same lane with it, there is a second dynamic obstacle O_2 which moves with velocity $v_{O_2} > v_{O_1}$. Since obstacle O_2 moves behind obstacle O_1, only obstacle O_1 is a visible obstacle (cf. Definition 3.3.3). The robot is accelerating, so that it can reach its maximum velocity and thus cover more ground on its journey towards its destination. Obstacle O_2 overtakes obstacle O_1. In such a situation, the hazard is that the robot does not activate any collision avoidance maneuvers, e.g., braking, due to very little to no reaction time. The potential effect of this hazard is a frontal crash with obstacle O_2, which represents an active collision caused by the robot with an obstacle from its environment.

Table 3.4.: Mobile Service Robot: Hazard Analysis and Risk Assessment.

ID	**Location**	**Phys. World Geometry**	**Environment**	**System Usage**	**System Behavior**	**Environment Behavior**	**Hazard**	**Potential Effect**
H1	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground}=0°$	*StraightRoad*, *DynamicObst*	*Driving towards a given destination*	*Accelerating*	*Dynamic obstacle overtakes a slower obstacle on the ego lane in front of the robot*	No robot's brake	Front crash possible
H2	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground}=0°$	*StraightRoad*, *DynamicObst*	*Driving towards a given destination*	*FullSpeedDrv*	*Dynamic obstacle accelerates on the ego lane in front of the robot*	No robot's brake	Front crash possible
H3	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground}=0°$	*StraightRoad*, *DynamicObst*	*Driving towards a given destination*	*FullSpeedDrv*	*Dynamic obstacle changes to the ego lane in front of the robot*	No robot's brake	Front crash possible
H4	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground}=0°$	*StraightRoad*, *DynamicObst*	*Driving towards a given destination*	*LeftLaneChange*	*Dynamic obstacle moving from behind to overtake the robot*	No robot's brake	Side crash possible
H5	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground}=0°$	*StraightRoad*, *DynamicObst*	*Driving towards a given destination*	*Braking*	*Dynamic obstacle moves in a blind spot behind the robot*	No robot's stop	Rear crash possible
H6	*Indoors*	$n_{lanes} \geq 3$, $\alpha_{Ground}=0°$	*StraightRoad*, *StationaryObst*	*Driving towards a given destination*	*Braking*	*Stationary obstacle starts to move on the ego lane in front of the robot*	No robot's stop	Front crash possible
...	...	...	...	...	...	...	...	...

Safety Properties

The HARA analysis carried out on the example of the mobile service robot has identified several hazards. For each of these hazards, the potential effect is a collision with an obstacle in the robot's environment actively caused by the robot. Safety requirements must be formulated for the robot in order to avoid these hazards and their effects on system and its environment.

Notice that a safety requirement can be formulated as formal safety property, since the intention of a safety requirement is to prevent the hazardous event over which it is formulated from ever happening. At a general level, a *safety property* informally states that "nothing bad ever happens". For the purpose of this example, the robot's safety property derived from the results of the HARA analysis is transcribed to the following informal specification in natural language:

Safety Property (Informal Specification). The robot must never actively collide with any obstacle in its environment.

The safety property is formally specified CTL. In CTL, one way to formulate a safety property is $AG\psi$. The operator A requires that a certain property is satisfied on all possible computation paths of the overall system model. The second operator G is the *Globally*-operator, which requests that a given property holds on the entire subsequent path. This is expressed in Equation (3.13) where ψ is the part contained inside the curly brackets.

$$\phi : AG\ [(y_R == y_{O_i}) \wedge (x_{O_i} - x_R \geq -c_R) \wedge (x_{O_i} - x_R \leq c_R) \rightarrow (v_R == 0)] \tag{3.13}$$

The property in Equation (3.13) translates to the robot being stationary, i.e., $v_R = 0$, at the moment when a visible obstacle, situated on the same lane as the robot, reaches the safety net of the robot. Once the obstacle is outside the borders of its safety net, the robot is considered to be out of danger and able to resume its journey towards its destination. A visual interpretation of the safety property is given in Figure 3.4.

Note that there is no reference made in the safety property with respect to the current velocity of the dynamic obstacle. Since the property is supposed to capture the desired behavior of the robot and not that of the dynamic obstacle, the latter is depicted in Figure 3.4 at the border of the robot safety net with a positive non-zero velocity. This is consistent with the system requirements in the functional system specification introduced in Section 3.3 of this chapter, namely that the movement of dynamic obstacles passing by the robot is emulated by their drive through the robot.

3.5. System Design

The system design of the mobile service robot is carried out on the basis of the system requirements in the functional system specification introduced in Section 3.3. Section 3.5.1 gives a short overview of how formal models are used in system desing to capture

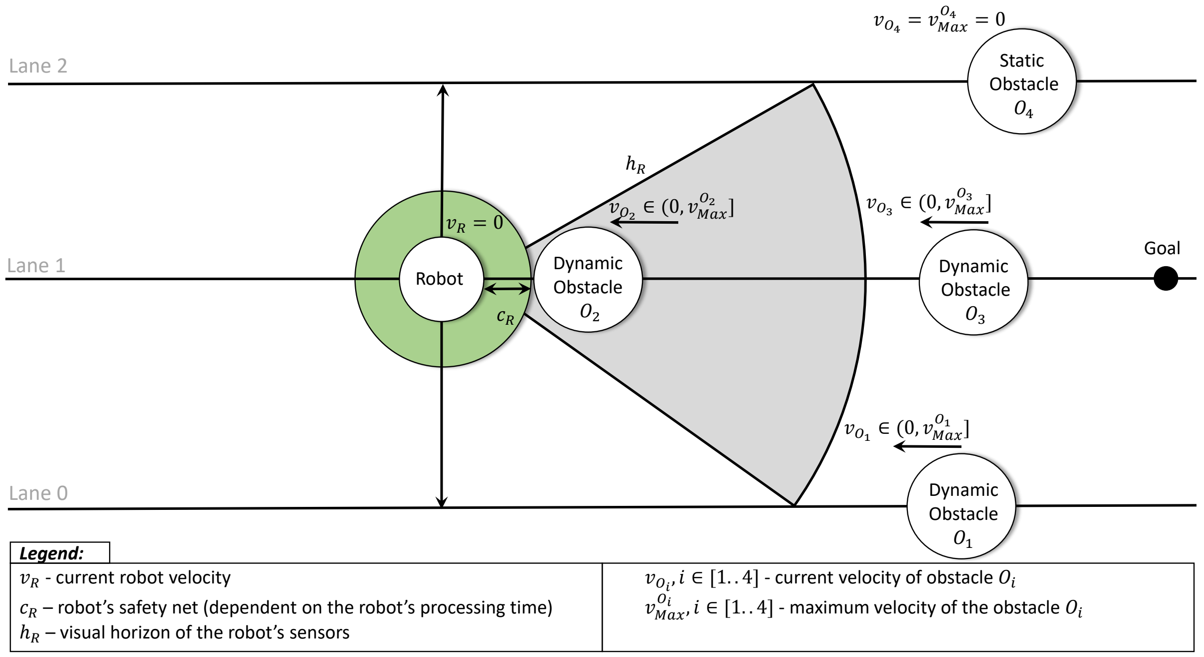

$v_{O_4} = v_{Max}^{O_4} = 0$
Static Obstacle O_4
Lane 2
h_R
$v_{O_2} \in (0, v_{Max}^{O_2}]$
$v_{O_3} \in (0, v_{Max}^{O_3}]$
$v_R = 0$
Goal
Lane 1
Robot
Dynamic Obstacle O_2
Dynamic Obstacle O_3
c_R
$v_{O_1} \in (0, v_{Max}^{O_1}]$
Lane 0
Dynamic Obstacle O_1
Legend:
v_R - current robot velocity
c_R – robot's safety net (dependent on the robot's processing time)
h_R – visual horizon of the robot's sensors
$v_{O_i}, i \in [1..4]$ - current velocity of obstacle O_i
$v_{Max}^{O_i}, i \in [1..4]$ - maximum velocity of the obstacle O_i

Figure 3.4.: Mobile Service Robot: A Visual Interpretation of the Safety Property.

different perspectives of the system under analysis. During the system design, a system model is created for the mobile service robot in Section 3.5.2, along with a model of the system environment, which is presented in Section 3.5.3. The system model of the mobile service robot and the environment model are combined in an overall system model, which is depicted in Section 3.5.4 along with the choices made for the modeling of the communication between the system and its environment.

3.5.1. Usage of Formal Models in System Design

Models rarely capture the entire system under study, since the size and complexity of most systems makes this an impossible task (cf. [SST18]). Therefore, a model is an abstraction of the system under study (cf. [Rod15]), which is built with an intended goal in mind, e.g., a mobile service robot tasked to drive autonomously towards a given destination. Although the model is a simplification of the actual system, it must be able to answer questions in place of the original system as if it were the system itself (cf. [JO01], [Rod15]). The fundamental question which must be answered is whether the system under study satisfies the system requirements defined in the functional system specification.

Different models can be developed for a system under study, which represent the system from different perspectives, e.g., external, interaction, structural or behavioral (cf. [Som14b]). An external perspective describes the context or the environment of the system, while an interaction perspective models the interactions between the system and its environment, or between the system components. The structural perspective allows the modeling of the inner structure of the system, as the system is refined into system functions through functional system design and into the respective software and hardware components during technical system design. In turn, a behavioral perspective models the dynamic behavior of the system and how it responds to events and external stimuli (cf. [Som14b]). The two models developed for the example of the mobile service robot comprise the external perspective, the interaction perspective as well as the behavioral perspective. The behavioral perspective is depicted in the system model, as the system model describes the desired behavior of the robot as specified in the system requirements. The environment model illustrates both the external and the behavioral perspective, since this model describes on an abstract level the environment of the mobile service robot as well as the dynamic behavior of the obstacles situated in the robot's environment. The interaction perspective models how the robot interacts with the obstacles in its environment, e.g., variables that emulate the robot's sensors observe obstacle features such as obstacle velocity.

Notice that the models created during system design have a double role. On one side, the design models can be used to communicate the requests of the software system architects on how the system is to be implemented. On the other side, these models can be verified against the system requirements in the functional system specification, in order to show whether the abstract representation of the system in interaction with its environment model satisfy the defined system requirements.

The system design process is usually carried out in an iterative manner, and the specific design activities can be interleaved with activities necessary to check whether the designed system model satisfies the system requirements. During the initial development iteration, the system designers develop a first version of the system model according to their understanding of the system requirements. The system model is then subject to reviews by experts, which can be accompanied further by other methods, e.g., simulation, testing, or formal verification, in an effort to obtain a system model which conforms to the system requirements. Should there be any inconsistencies in the system model with respect to the system requirements, the system designer can revisit the designed models and make the necessary adjustments. The process is repeated until a system model is obtained, which satisfies the defined system requirements.

Formal verification methods can be used at design-time to check whether the developed system model satisfies the specified system requirements. In case the system designers decide to use formal verification, it can be beneficial to design the system model and the environment model with verification in mind, in order to facilitate the usage of design-time verification methods on these models.

3.5.2. Environment Model

In the example of the mobile service robot, two models are created during the system design phase, a model of the autonomous system and a model of its environment. In an effort to create a system model in accordance with the system requirements, these models are created with formal verification in mind, which is carried out also in the system design phase.

The autonomous system and its environment in the motivational example are modeled as two distinct Uppaal timed automata, which are an extension of the timed automata formalism introduced by Alur and Dill in [AD94]. See Section 2.4.1 in Chapter 2 for the main definitions with respect to the notion of *timed automata*. For the sake of brevity and without impending the full description of the example, both models are depicted only through abstract representations of their corresponding timed automata.

The environment model describes the behavior of the obstacles in the system's environment and is illustrated in Figure 3.5. This automaton is henceforth referred to as the *obstacle automaton*.

In order to model the robot's environment illustrated in Figure 3.1, four instances of the obstacle automaton are necessary, i.e., three for modeling the respective dynamic obstacles and the forth for describing the stationary obstacle. In order to create several instances of it, an automaton template modeled in the Uppaal model checker must be provided with a set of parameters. In the motivational example, several parameters are defined for the Uppaal template corresponding to the obstacle automaton: identification number of the obstacle, position of the obstacle on its lane, and a boolean flag which models whether the obstacle is stationary or not.

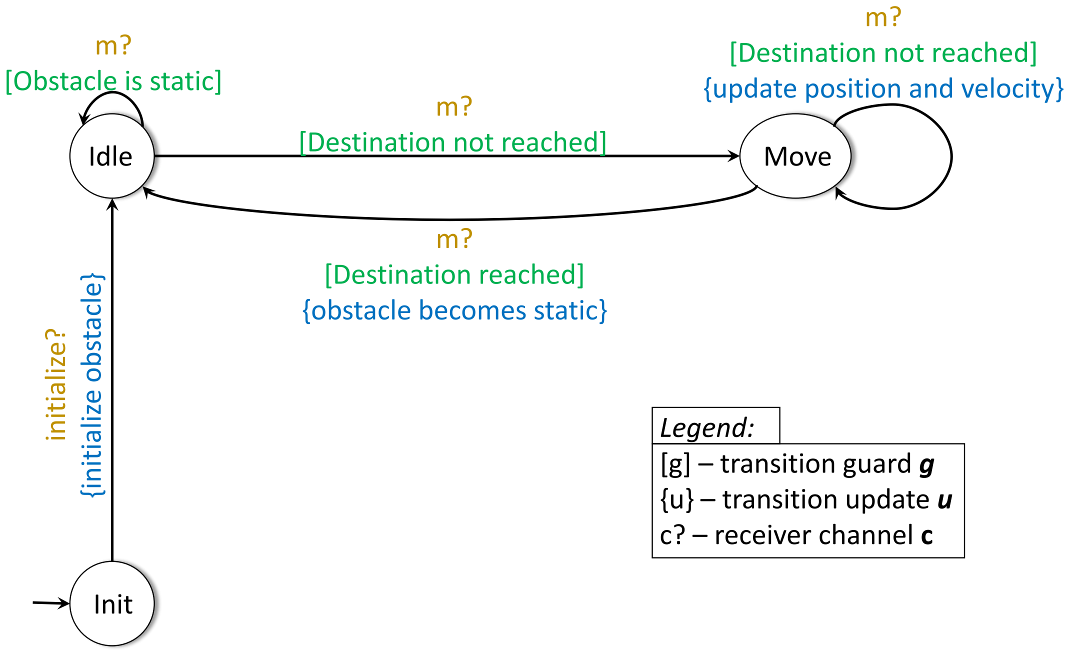


Figure 3.5.: Mobile Service Robot: Environment Model.

The obstacle automaton has three locations, the initial location `Init` and the locations `Idle` and `Move`. The transition from `Init` to `Idle` serves as an initialization step for the obstacles, i.e., the parameters of each obstacle are initialized with their respective values and are stored in a shared data structure. When in `Idle`, an obstacle can be either stationary or it can start to move towards the robot in a continuous motion. Assuming that there are n obstacles in the robot's environment, remember that each obstacle

$O_i, i \in [1..n]$ has a specific maximum bound for its current velocity, which is denoted $v_{Max}^{O_i}$. For a stationary obstacle, the corresponding instance of the obstacle automaton remains in the location `Idle` and its maximum velocity is considered to be null. In the case of a dynamic obstacle, the corresponding automaton instance enters the location `Move` and chooses arbitrarily from the interval $(0, v_{Max}^{O_i}]$ a value for the current velocity each time the dynamic obstacle O_i executes a move. Based on its current velocity, the new position of the obstacle O_i is computed and updated in the shared data structure. The automaton is modeled so that each moving obstacle has a destination, and upon reaching it, the obstacle comes to a full stop, i.e., the automaton enters the location `Idle`. This behavior ensures that the dynamic obstacles will eventually stop moving and become stationary.

3.5.3. Technical System Model

The technical system model, also called the *robot automaton*, is depicted in Figure 3.6 and describes the robot's behavior. Initially, the automaton is in its initial location, `Idle`, as the robot has not yet started to move. At the same time, `Idle` is also the final location, in which the automaton enters when the robot reaches its destination. As long as the robot has not reached its maximum velocity, the automaton remains in the location `Accelerate`. The robot drives with maximum velocity, as long as no collision danger is detected, i.e., the automaton enters the location `Drive` and remains there for as long as possible. Should the robot detect a collision danger or should it approach its destination, then the automaton enters the location `Brake`, which enables the robot to reduce its velocity, and eventually, come to a standstill (location `Stop`).

Nevertheless, before braking to a standstill, the robot checks to see whether it can drive further, albeit with a reduced velocity. This is can be justified by the inherent purpose of mobile service robots, i.e., to autonomously reach their destination and to cover as much ground as possible in the process before having to brake. While the robot is braking, it may happen that another lane becomes safe to move on. Remember that a lane is considered to be safe if no visible obstacles are detected inside the region spanned by the robot's collision distance. The robot can use the opportunity of changing to a safe lane in order to drive farther rather than trigger the emergency brake. However, if no lane change is possible, then the collision danger persists and the robot is forced to brake until it comes to a full stop, i.e., the automaton enters the location `Stop`.

3.5.4. Overall System Model

In order to perform the computations for the collision distance, the robot needs to know the current position of the obstacle and its current velocity, both computable from the received sensor data. In order to emulate the robot's sensors, the robot automaton communicates with the obstacle automaton using shared variables. In Uppaal, this kind of variables are modeled with the help of global variable declarations. Figure 3.7 illustrates the section of global declarations in the Uppaal project created for the modeling of the motivational example.

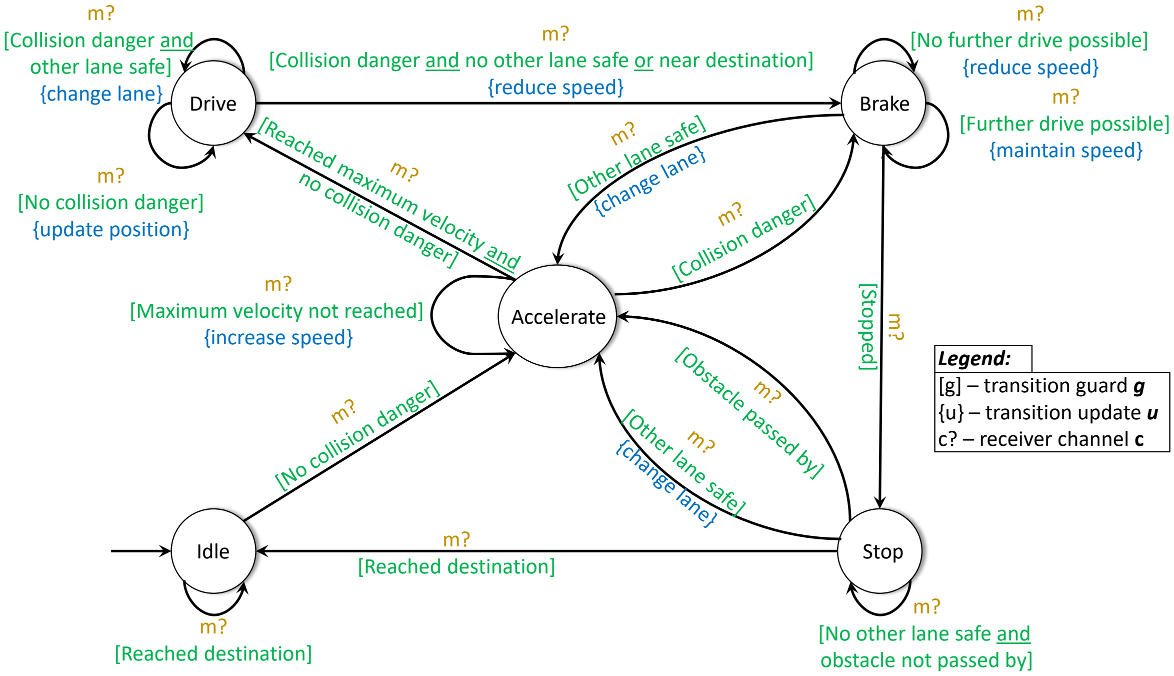


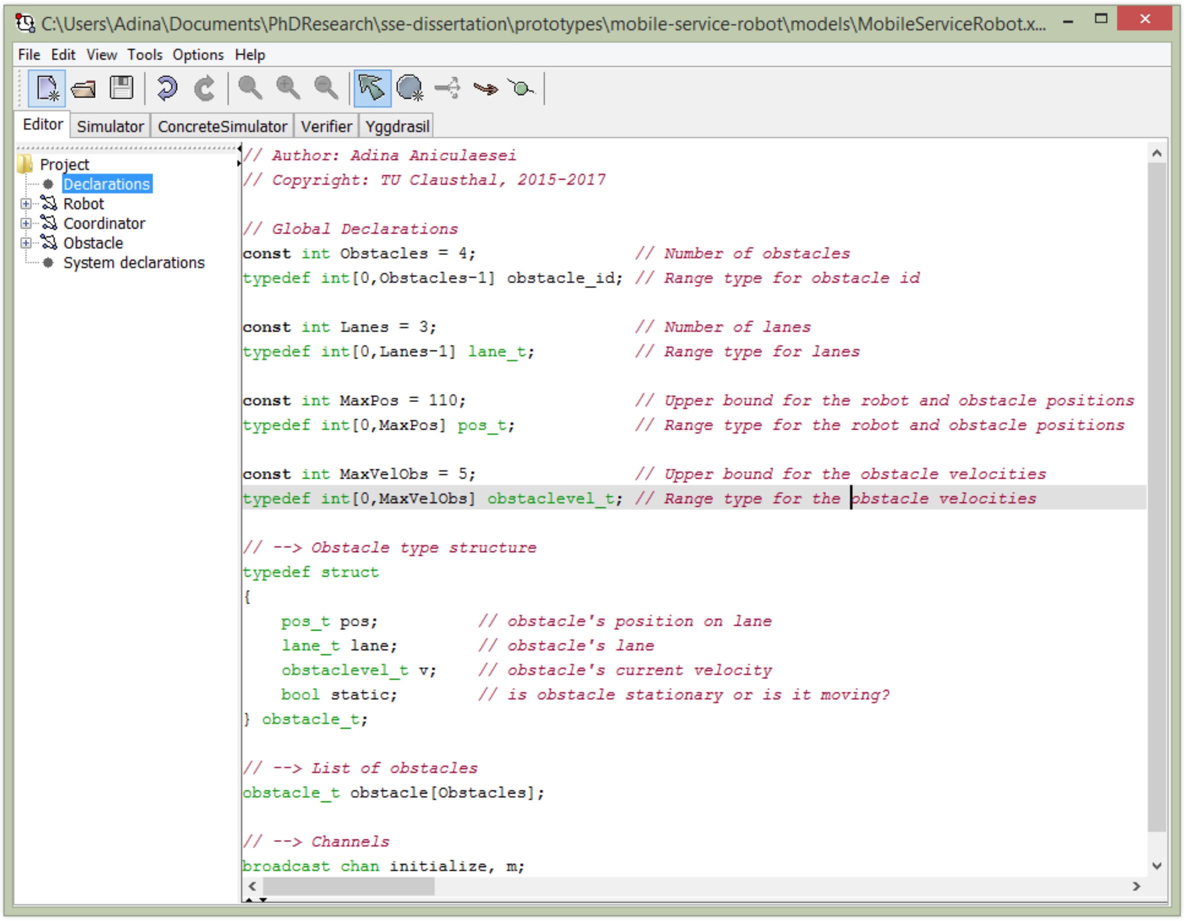
Figure 3.6.: Mobile Service Robot: Technical System Model.

Note that the structure `obstacle_t` defines a type which contains all the information which the robot must know about an obstacle: its coordinates in the environment, its current velocity and its current status. Each time a dynamic obstacle moves, the respective instance of the obstacle automaton updates the obstacle's current position and velocity in the corresponding element of the list `obstacle`. The information stored in this list emulates the sensor data, which is received by the robot and used further in its processing step. As mentioned in Section 3.5.2, the obstacle automaton is modeled so that dynamic obstacles have a destination, which in this example is the point of origin of the Cartesian coordinate system. Upon reaching this point, dynamic obstacles become stationary. This means that, in addition to its position and velocity, the status of each dynamic obstacle must also be updated in the shared list of obstacles.

As a side note, each numerical variable in the obstacle type structure has its value domain restricted by an upper bound. During the creation of the overall system model, restriction of variables domains is applied in order to decrease the size of the model. While the restriction of the variable domain is a deliberate decision taken during modeling to reduce the size of the model's state space, Uppaal supports further techniques for dealing with the state space explosion problem (cf. Section 2.4.1).

As an autonomous system, the robot performs during its runtime operation three specific activities:

- *sensing* - capturing the runtime environment through its sensors,
- *processing* - reasoning on the basis of the received sensor data and planing of the corresponding actions, and
- *acting* - execution of the planned actions.


C:\Users\Adina\Documents\PhDResearch\sse-dissertation\prototypes\mobile-service-robot\models\MobileServiceRobot.x...
File Edit View Tools Options Help
Editor
Simulator
ConcreteSimulator
Verifier
Yggdrasil
Project
Declarations
Robot
Coordinator
Obstacle
System declarations

```
// Author: Adina Aniculaesei
// Copyright: TU Clausthal, 2015-2017

// Global Declarations
const int Obstacles = 4;                // Number of obstacles
typedef int[0,Obstacles-1] obstacle_id; // Range type for obstacle id

const int Lanes = 3;                    // Number of lanes
typedef int[0,Lanes-1] lane_t;          // Range type for lanes

const int MaxPos = 110;                 // Upper bound for the robot and obstacle positions
typedef int[0,MaxPos] pos_t;            // Range type for the robot and obstacle positions

const int MaxVelObs = 5;                // Upper bound for the obstacle velocities
typedef int[0,MaxVelObs] obstaclevel_t; // Range type for the obstacle velocities

// --> Obstacle type structure
typedef struct
{
    pos_t pos;          // obstacle's position on lane
    lane_t lane;        // obstacle's lane
    obstaclevel_t v;    // obstacle's current velocity
    bool static;        // is obstacle stationary or is it moving?
} obstacle_t;

// --> List of obstacles
obstacle_t obstacle[Obstacles];

// --> Channels
broadcast chan initialize, m;
```

Figure 3.7.: Mobile Service Robot: Declarations of Shared Variables in Uppaal.

From this general description of the robot's behavior, it follows that the actions performed in the acting phase are a consequence of the received sensor data, which record the changes occurring in the environment. On one hand, from the perspective of the concrete actions executed in acting phase, i.e., braking in case of collision danger, it can be considered that the environment influences the behavior of the robot. On the other hand, seen at an abstract level, the robot performs the activities intrinsic to an autonomous system independently of the changes taking place in its environment. Regarded at the same abstract level, the activities performed by the robot do not exert any influence on what changes occur in its environment.

In order to model the robot and its environment as two entities acting independently of each other, a coordinator is added to the overall system model in order to overview the communication between the robot automaton and the obstacle automaton and ensure fairness between them. Similarly to the robot and its environment, the coordinator is described using Uppaal timed automata. For the sake of consistency with the depiction of the other two automata, Figure 3.8 illustrates not the actual Uppaal timed automaton modeling the coordinator, but an abstract representation of it.

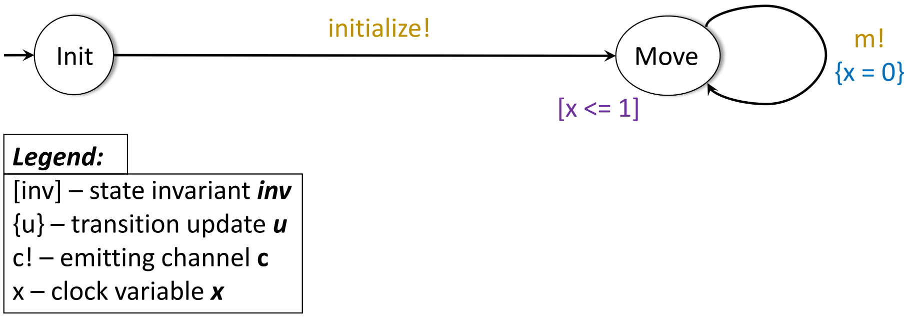


Figure 3.8.: Mobile Service Robot: Coordinator Model in the Motivational Example.

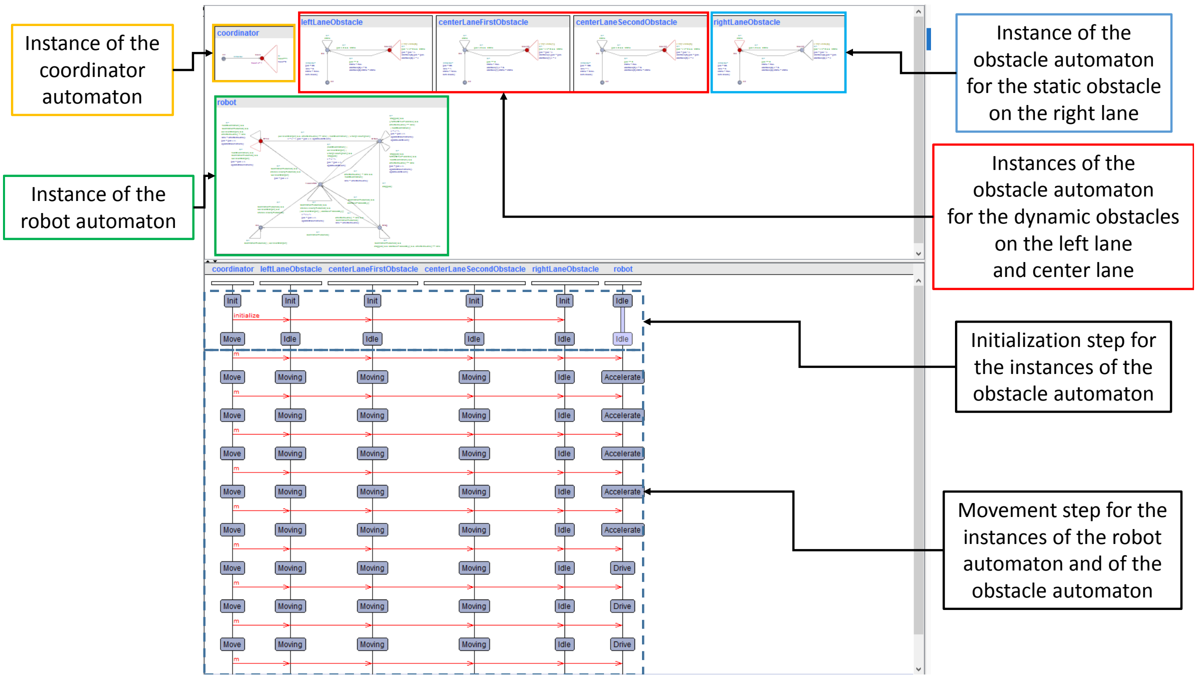
The coordinator automaton has two locations, the initial location `Init` and the location `Move`. On the transition from `Init` to `Move`, the coordinator automaton sends an initialization command to all the instances of the obstacle automaton via the communication channel `initialize`. Upon receiving this command, each instance of the obstacle automaton is initialized with the corresponding parameter values and the information is stored in the shared obstacle list. Through this step, it is ensured that the information about the initial state of the environment becomes available for the robot, before the robot executes any move. This is justified by the robot's necessity to have access to initial information about its environment in order to be able to plan its route towards its destination.

Following the initialization step, the coordinator automaton sends movement commands to the robot automaton as well as to all the instances of the obstacle automaton. According to the environment model in Section 3.5.2, the movement command sent to the obstacle automaton leads to different behaviors, depending on the status of each obstacle in the environment. Thus, stationary obstacles are required to remain idle (location `Idle` in Figure 3.5), while dynamic obstacles are requested to move forward with random velocity (location `Move` in Figure 3.5). With each executed move, an instance of the obstacle automaton updates the position, velocity and status information of the corresponding obstacle in the shared obstacle list.

Notice that the coordinator model also defines a clock, modeled by the variable $x \in \mathbb{R}_{\geq 0} \cap [0, 1]$. The motivation for the clock variable is two-folded. On one hand, it helps to ensure fairness between the technical system model and the environment model, as neither the robot automaton nor the obstacle automaton are allowed to spend more than one time unit for the execution of a transition. On the other hand, the number of the symbolic states of the overall system model computed by Uppaal during model checking is drastically reduced by the bound set on the clock variable.

The coordinator automaton sends its commands to the robot and obstacle automata via broadcast channels. This type of channels allows 1-to-many synchronizations. The intuition behind broadcast channels is the following: an edge with synchronization label `e!` emits a broadcast on the channel `e` and any enabled edge with synchronization label `e?` will synchronize with the emitting automaton. This means that an edge which emits on a broadcast channel can always fire, provided that its guard is satisfied. The update on the emitting edge is executed first. In the coordinator automaton, this update is represented by the clock reset on the edge sending on channel `m`. The update on the receiving edges are executed left-to-right in the order in which the processes are given in the system definition. In this way, before its every move, the robot automaton has access to the latest information about the obstacles in its environment.

The overall system model, composed of the coordinator automaton, the robot automaton and four instances of the obstacle automaton, is depicted in the upper part of Figure 3.9. In the lower part of Figure 3.9, a schematic sequence diagram illustrates the communication between the components of the overall system model. The parallel composition and the communication between Uppaal automata is realized through shared variables and synchronization channels (cf. Section 2.4.1).



Instance of the coordinator automaton

Instance of the robot automaton

Instance of the obstacle automaton for the static obstacle on the right lane

Instances of the obstacle automaton for the dynamic obstacles on the left lane and center lane

Initialization step for the instances of the obstacle automaton

Movement step for the instances of the robot automaton and of the obstacle automaton

Figure 3.9.: Mobile Service Robot: Overall System Model and Communication between its Components in the Simulator Panel of Uppaal.

3.5.5. Specification of System Properties

Both the safety property as well as the liveness property of the mobile service robot are formalized in TCTL, which is the specification language recognized by the Uppaal model checker. The formal language TCTL is a real-timed variant of CTL, created to express properties about timed automata (cf. Section 2.3.1). As such, several operators are similar to those introduced by CTL. For example, the operator A retains its semantic from CTL, by which it is required that a certain property be satisfied on all possible execution paths of the overall system model. The operator $\Box$ has the same semantic as the *Globally*-operator known from the CTL language, by which it requires that a given property holds on the entire subsequent path (cf. Section 2.3.1).

The safety property introduced in Equation (3.13) is translated in TCTL in Equation (3.14).

$$\begin{aligned}\phi : A\Box \; \{\mathit{forall}\ (i : \mathit{int}[0, n-1])\ (\mathit{robot.lane} \;==\; \mathit{obstacles}[i].\mathit{lane})\\ \mathit{and}\ (\mathit{obstacles}[i].\mathit{pos} \;-\; \mathit{robot.pos} \;\geq\; -\mathit{robot.c})\\ \mathit{and}\ (\mathit{obstacles}[i].\mathit{pos} \;-\; \mathit{robot.pos} \;\leq\; \mathit{robot.c})\\ \mathit{imply}\ (\mathit{robot.v} \;==\; 0)\}\end{aligned} \tag{3.14}$$

In a similar manner to its safety property, the liveness property of the mobile service robot is also expressed in TCTL. The operator A carries the same semantic as in the case of the safety property. The operator $\Diamond$ is the Uppaal specific notation for the operator $F(finally)$, which requires that a given property is eventually satisfied (cf. Section 2.3.1). This means that, on all computation paths of the overall system model, there is a state that satisfies the given property and which is eventually reached. The TCTL formula in Equation 3.15 corresponds to the CTL liveness property from Equation 3.12.

$$\theta : A\Diamond(\mathit{robot.pos} - \; \mathit{robot.goalPos} \;\geq\; 0) \tag{3.15}$$

Notation 3.5.1. The following correspondence is defined between the notations used in Equations 3.14 and 3.15 and those introduced in Notation 3.1.1:

1. $(\mathit{robot.pos}, \mathit{robot.lane}) \stackrel{\text{def}}{=} (x_R, y_R)$,
2. $(\mathit{robot.goalPos}, \mathit{robot.goalLane}) \stackrel{\text{def}}{=} (x^R_{Goal}, y^R_{Goal})$,
3. $\mathit{robot.c} \stackrel{\text{def}}{=} c_R$,
4. $\mathit{robot.v} \stackrel{\text{def}}{=} v_R$, and
5. $(\mathit{obstaacle}[i].\mathit{pos}, \mathit{obstacle}[i].\mathit{lane}) \stackrel{\text{def}}{=} (x_{O_i}, y_{O_i})$.

■

3.5.6. Design-Time Verification

For model checking, the overall system model is built as network timed automaton through the parallel composition of all the instances of the technical system model, of the environment model and of the coordinator model. See Chapter 2 for a general definition of the parallel composition of timed automata. According to this definition, the set of states of the network automaton is the Cartesian product of the sets of states of all instances of its component automata.

The first step in illustrating the application of model checking on the motivational example is to discuss which state variables play a role in the modeling of the robot automaton and which in that of the obstacle automaton.

The state of the robot automaton is defined by

$$s_R = (x_R, y_R, v_R, v^R_{Max}, x^R_{Goal}, y^R_{Goal}, h_R, c_R) \tag{3.16}$$

in which h_R is the visual horizon of the robot's sensors and c_R is its safety net. For all the other state variables, the notations introduced in Section 3.1 apply.

In a similar manner, the state of the obstacle automaton instance corresponding to an obstacle O_i is defined by

$$s_{O_i} = (x_{O_i}, y_{O_i}, v_{O_i}, v_{Max}^{O_i}, \mathit{isStatic}_{O_i}), i \in [1..n] \tag{3.17}$$

where $n \in \mathbb{N}_{>0}$ is the number of obstacles, while $\mathit{isStatic}_{O_i}$ is a flag which is *true* if the obstacle O_i is stationary and *false* otherwise. For the other state variables, the notations established in Section 3.1 are valid.

The verification of the network automaton against a given property specification follows a straightforward approach. The UPPAAL model checker performs a reachability analysis, in which it computes the set of reachable states for the network automaton. During this analysis, the set of reachable states is spanned into a tree, with the initial state of the network automaton as the root of the tree and the reachable states as its nodes. The model checker examines each node of the tree and checks whether the given property is satisfied or not. In case the model checker finds a state which violates the given property, then it provides not only the verification result but also a counterexample, which consists of the path in the spanning tree that leads to the respective state. This approach is applied for the verification of both properties in Equation (3.13) and in Equation (3.15).

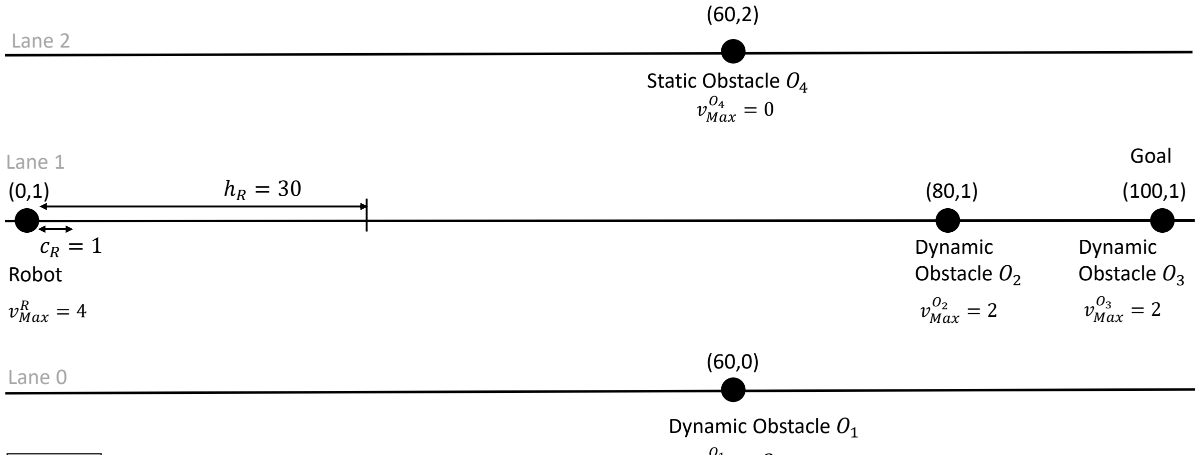
Several of the state variables are used to model the collision avoidance algorithm implemented in the robot automaton. Some of these variables are present also in the specification of the safety property. For the sake of brevity, the representation of the network automaton states in the spanned tree uses only the state variables which are used to formulate the specification of the safety property. Thus, the states of the network automaton displayed in the spanning tree consist of the robot coordinates, the coordinates of the obstacles in the robot's environment as well as the robot's velocity and its safety net. Each state of the network automaton is denoted by the tuple

$$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), \cdots, (x_{O_n}, y_{O_n}), v_R, c_R) \tag{3.18}$$

where $n \in \mathbb{N}_{>0}$ is the number of obstacles.

The motivational example consists of a mobile robot and its environment with three dynamic obstacles and one stationary obstacle. A visual representation of the initial state of the corresponding network automaton is given in Figure 3.10. Initially, both the robot and the dynamic obstacles are stationary, i.e., $v_R = 0$ and $v_{O_i} = 0, \forall i \in [1..n]$. The maximum velocity of the robot is $v_{Max}^R = 4$, while the dynamic obstacles $O_i, i \in \{1, 2, 3\}$ have, for the sake of simplicity, the same maximum velocity $v_{Max}^{O_i} = 2$. The visual horizon of the robot's sensors is set at $h_R = 30$ distance units. In order to simplify the analysis, the safety net of the robot is considered to be constant and one distance unit wide, i.e., $c_R = 1$, although in reality it is a function of the current robot velocity v_R. According to its visual representation depicted in Figure 3.10, the initial state of the network automaton can be represented as $s_0 = ((0, 1), (60, 0), (80, 1), (100, 1), (60, 2), 0, 1)$.

In Figure 3.11 the inner details of the verification of the robot's behavior with respect to its safety property are illustrated. The path highlighted in the spanning tree shows the brake maneuver, starting in state $((48, 1), (40, 0), (58, 1), (78, 1), (60, 2), 3, 1)$, which



Legend:

(0,1) – initial position of the robot	(60,0) – initial position of the dynamic obstacle O_1
(100,1) – goal coordinates	(80,1) – initial position of the dynamic obstacle O_2
h_R – visual range of the robot’s sensors	(100,1) – initial position of the dynamic obstacle O_3
c_R – robot’s safety net (dependent on the robot’s processing time)	(60,2) – initial position of the static obstacle O_4
v_{Max}^{R} - maximum velocity specific to the robot	$v_{Max}^{O_i}$ - maximum velocity specific to obstacle O_i, $i \in [1..4]$

Figure 3.10.: Mobile Service Robot: A Visual Representation of the Network Automaton’s Initial State in Uppaal.

the robot triggers because it perceives the dynamic obstacle O_2 as a collision danger in its environment. Notice that the obstacle O_3 is situated further behind the obstacle O_2, and thus invisible to the robot, even if it has already entered its sensor horizon.

When the obstacle O_2 reaches its safety net, the robot has already come to a full stop, i.e., the network automaton enters the state $((51,1),(35,0),(52,1),(74,1),(60,2),0,1)$. Once the obstacle O_2 has left its safety net, the robot resumes its drive as illustrated in the state $((52,0),(32,0),(48,1),(71,1),(60,2),1,1)$ of the network automaton. Notice that in this state, the robot chooses to change the lane rather than to accelerate to its maximum velocity only to be forced to brake later due to the collision danger posed by the dynamic obstacle O_3. The only safe lane is L_0, since O_1 has already driven by the robot and the lanes L_1 and L_2 are occupied by the obstacles O_3 and O_4 respectively.

There is one state in the network automaton in which the robot always satisfies its safety property, namely when the robot is at rest in its initial position. However, the robot is an autonomous system which is assigned a task that can be completed only by driving towards and reaching a given destination. Therefore, it is important to show that the robot satisfies its safety property while fulfilling its assignment. The fact that the robot completes its assignment can be proven by the verification of the liveness property in Equation (3.15). In Figure 3.12, a view of the verifier panel in the graphical user interface of the Uppaal model checker displays the verification result for the safety property and the liveness property, i.e., both are satisfied.

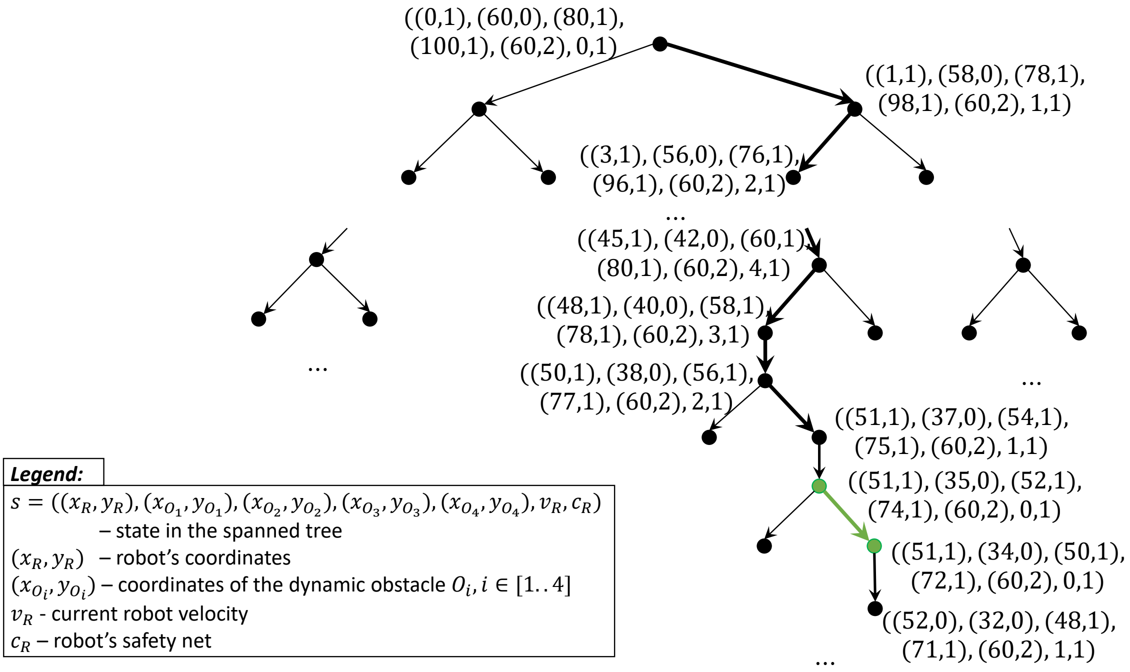
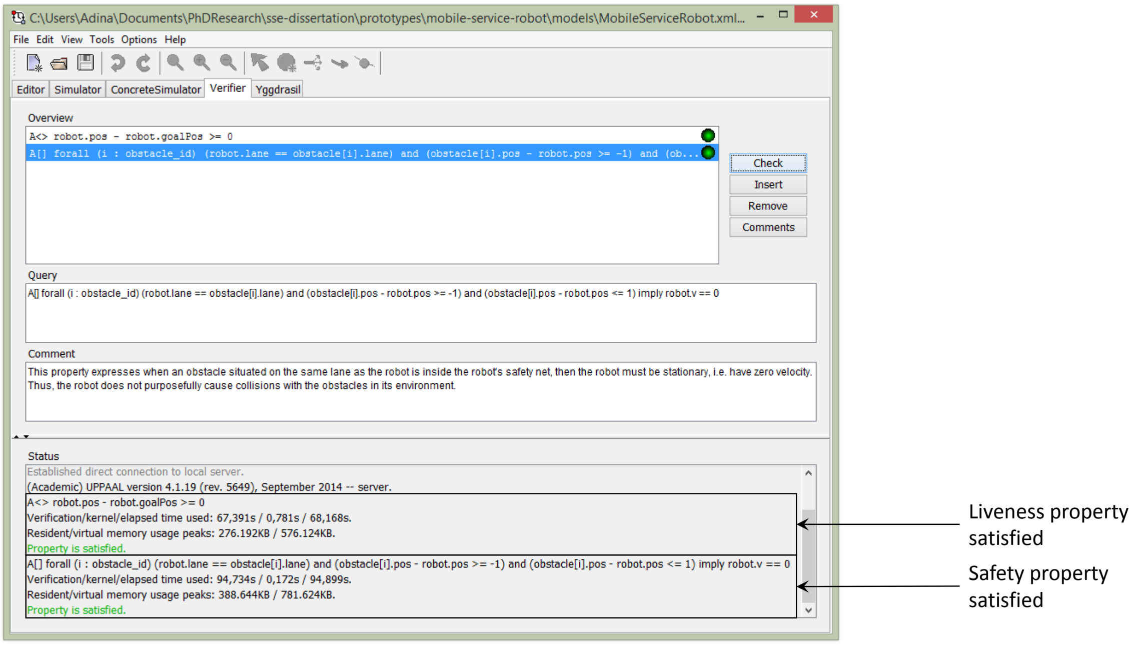


Figure 3.11.: Mobile Service Robot: Spanning Tree Path Showing its Satisfied Safety Property.

3.6. System Implementation

The system implementation consists of the implementation of the functional and technical system architecture. The systems functions defined in the functional system architecture are realized mainly through software components which in turn make up the software architecture of the system. Depending on the way in which the system development process is organized, a part of software components can be developed by the system manufacturer while the development of the rest of the software components is outsourced to multiple suppliers of software solutions. By comparison, hardware components are not constructed by the system manufacturer, but are acquired from multiple vendors, e.g., laser scanner or infrared sensors for autonomous robots.

Each software component is implemented in an agreed upon programming language, e.g., C/C++. The source code of the software components can be manually written or generated from models, e.g., finite state machines [KJ10, HMU14], Statecharts [Har87] or signal flow graphs [FPEN15]. Code generation from models is in fact model-to-text (M2T) transformation, which is a key of model-driven development (cf. [Rod15]). There are various commercial toolchains which support model-based software development, e.g Matlab/Simulink [BD97], Ascet-sd [LBBZ97], or Ansys Scade [CPP17]. The code generators included in these toolchains ensure that the M2T transformation is performed correctly, and that the generated code is a correct implementation of the model. In addition to code generation, these toolchains allow the simulation of the designed model. The system engineers can via simulation try out different scenarios with various system


C:\Users\Adina\Documents\PhDResearch\sse-dissertation\prototypes\mobile-service-robot\models\MobileServiceRobot.xml...
File Edit View Tools Options Help
Editor
Simulator
ConcreteSimulator
Verifier
Yggdrasil
Overview
Check
Insert
Remove
Comments
Query
Comment
Status
Liveness property satisfied
Safety property satisfied

```
A<> robot.pos - robot.goalPos >= 0
A[] forall (i : obstacle_id) (robot.lane == obstacle[i].lane) and (obstacle[i].pos - robot.pos >= -1) and (ob...
```

Query: A[] forall (i : obstacle_id) (robot.lane == obstacle[i].lane) and (obstacle[i].pos - robot.pos >= -1) and (obstacle[i].pos - robot.pos <= 1) imply robot.v == 0

Comment: This property expresses when an obstacle situated on the same lane as the robot is inside the robot's safety net, then the robot must be stationary, i.e. have zero velocity. Thus, the robot does not purposefully cause collisions with the obstacles in its environment.

```
Established direct connection to local server.
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
A<> robot.pos - robot.goalPos >= 0
Verification/kernel/elapsed time used: 67,391s / 0,781s / 68,168s.
Resident/virtual memory usage peaks: 276.192KB / 576.124KB.
Property is satisfied.
A[] forall (i : obstacle_id) (robot.lane == obstacle[i].lane) and (obstacle[i].pos - robot.pos >= -1) and (obstacle[i].pos - robot.pos <= 1) imply robot.v == 0
Verification/kernel/elapsed time used: 94,734s / 0,172s / 94,899s.
Resident/virtual memory usage peaks: 388.644KB / 781.624KB.
Property is satisfied.
```

Figure 3.12.: Mobile Service Robot: Safety Property and Liveness Property Visualized in Uppaal.

parameter configurations and can manipulate the system environment model in order to see whether the designed system model conforms to the specified system requirements.

No implementation is carried out for the models defined in the example of the mobile service robot as they are described in Section 3.5. Instead, the facility of simulation provided by the Uppaal model checker has been used in order to analyze the system model with respect to the defined system requirements.

3.7. System Test

Testing is the next phase in the development process, following the system's implementation. During testing, the test object is subject to various stimuli and the result of its computations is compared to the expected values defined by the test engineers (cf. Section 2.5.1). With the exception of the system component test, where software components are individually tested, the test object is in all the other test phases tested in interaction with its environment. Depending on the hierarchy level at which the test object is situated in the system, its environment consists of a technical environment and/or a physical environment (cf. Section 2.5.3). Notice that, since irrespective of its granularity, any test object has an environment, the approach for system testing described in this section can be applied at any defined system level.

In the example of the mobile service robot, the overall system model consisting of the technical system model and the environment model (cf. Section 3.5) is verified against the robot's safety requirement using model checking. In order to carry out the system test, tests have to be constructed for the mobile service robot. Remember that the system test is meant to check whether the system as a whole conforms to the defined system requirements (cf. Section 3.2). Model checking is a method for systematic generation of test cases by building trap properties from formal requirements (cf. Section 2.5.1). This method has been firstly applied in previous research in aeronautics domain (cf. [WRHM06], [SWRH10]) and subsequently transferred to automotive control systems (cf. [AHDR18]). The counterexample generated as a result of model checking the system against the trap property provides the test inputs for a test case which satisfies the original system requirement.

Besides test inputs, a test case contains also preconditions and postconditions for the test case's execution and the expected output (cf. Section 2.5.1). Checking whether a system S passes a given test case tc can be done with the help a test oracle. There are various methods to create test oracles. One of them is to derive test oracles as property monitors from the system requirements themselves. (cf. Section 2.5.1). In the example of the mobile service robot, a test oracle has been derived from its safety property in the form of a property monitor.

Notice that, during the execution of a test case, the property monitor observes the system execution which has served as basis for the respective test case and compares in each state of the system trace the result of system execution with the expected result. The system satisfies its property specification if the result of the system execution and the expected result coincide in every state of the observed system execution. This can be expressed in predicate formalisms over system traces, e.g., in temporal logic such as LTL. In LTL, safety properties can be expressed using the formula $G\ \psi$. The safety property for the motivational example, denoted by ϕ, is illustrated in its LTL form in Equation (3.19), where ψ is the part contained in the curly brackets.

$$\phi : G\{(y_R\ ==\ y_O) \wedge (x_O - x_R \geq -c_R) \wedge (x_O - x_R \leq c_R) \rightarrow (v_R == 0)\} \tag{3.19}$$

Note that in Equation 3.19 the pair (x_O, y_O) denotes the current coordinates of a visible obstacle in the robot's environment.

In order to derive a monitor from the robot's safety property ϕ, the conditional statement in the curly brackets must be transformed in a well-formed formula composed of several atomic predicates connected by logical conjunction and disjunction. For this purpose, the safety property is first broken down into the following predicates:

- $p_1 : y_R == y_O$
- $p_2 : x_O - x_R \geq -c_R$
- $p_3 : x_O - x_R \leq c_R$
- $p_4 : v_R == 0$

Secondly, to transform the conditional statement the material implication rule defined in propositional logic is used. This rule, written in the sequent notation, is given in Equation 3.20.

$$(P \rightarrow Q) \vdash (\neg P \vee Q) \tag{3.20}$$

By applying the material implication rule, the conditional statement in the runtime safety property ϕ is substituted by a disjunction in which the antecedent is negated. In this way, the property monitor M_ϕ, depicted in Equation 3.21, is derived from the safety property ϕ.

$$M_\phi : \neg(p_1 \wedge p_2 \wedge p_3) \vee p_4 \tag{3.21}$$

The property monitor M_ϕ is then evaluated in each state of a system trace and yields a certain verdict. If there is a state in the system trace for which the property monitor M_ϕ returns *false*, then the robot's behavior in the runtime environment violates the safety property ϕ.

For the sake of consistency with the design-time verification, the notation for the system states is reused for the system test of the mobile service robot. Thus, a state in a system execution of the robot is denoted by the same tuple $s = ((x_R, y_R), (x_{O_1}, y_{O_1}), \cdots, (x_{O_n}, y_{O_n}), v_R, c_R)$, with $n \in \mathbb{N}_{>0}$ as the number of obstacles. The values of the state variables in the initial state of the network automaton are used as initial values for the system test of the robot. In order to simplify the analysis during system test, the safety net of the robot is set to be constant and one distance wide, i.e., $c_R = 1$, although in reality it varies with the current robot velocity v_R. Thus, the initial state for any finite system trace is represented, as in the design-time verification, by $s_0 = ((0, 1), (60, 0), (80, 1), (100, 1), (60, 2), 0, 1)$. The visual representation of the initial system state used at design-time remains valid also for system test.

Before illustrating the system test of the safety property ϕ, several definitions and notations regarding the evaluation of the property monitor M_ϕ must be introduced.

Definition 3.7.1 - Variable Interpretation
Let $\mathcal{X}$ be a finite set of state variables taking values in $\mathbb{R}_{\geq 0}$. An interpretation ν over $\mathcal{X}$ is a function $\nu : \mathcal{X} \to \mathbb{R}_{\geq 0}$ which associates each variable $x \in \mathcal{X}$ with its concrete value $\nu(x) \in \mathbb{R}_{\geq 0}$. ■

Notation 3.7.1. In order to define the evaluation of the property monitor M_ϕ, the following notations are established.

1. For any obstacle $O_i, i \in [1..n]$, $O \mapsto O_i$ is a mapping relation by which the variables $x_O, y_O \in \mathcal{X}$ are evaluated to the values of the variables $x_{O_i}, y_{O_i} \in \mathcal{X}$ respectively. This is denoted by $O \mapsto O_i \overset{\text{def}}{=} \{\nu(x_O) = \nu(x_{O_i}), \nu(y_O) = \nu(y_{O_i})\}$.
2. The evaluation of a boolean predicate p with the coordinates of an obstacle O_i, $i \in [1..n]$ is denoted by $p[O \mapsto O_i]$.

■

Definition 3.7.2 - Evaluation of the Property Monitor M_ϕ
Let $n \in \mathbb{N}_{>0}$ be the number of obstacles in the robot's environment. The evaluation of the property monitor M_ϕ for the property specification ϕ with the coordinates of an obstacle O_i, $i \in [1..n]$ is defined as

$$M_\phi[O \mapsto O_i] \overset{\text{def}}{=} \neg(p_1[O \mapsto O_i] \wedge p_2[O \mapsto O_i] \wedge p_3[O \mapsto O_i]) \vee p_4$$

■

In Figure 3.13, a system trace of the robot as well as the corresponding monitor trace are depicted. Table 3.5 presents in detail the evaluation of the safety property monitor M_ϕ in each state illustrated in the system trace. The evaluation of the property monitor in each state of the system execution considers only those obstacles which are visible to the robot in that respective state.

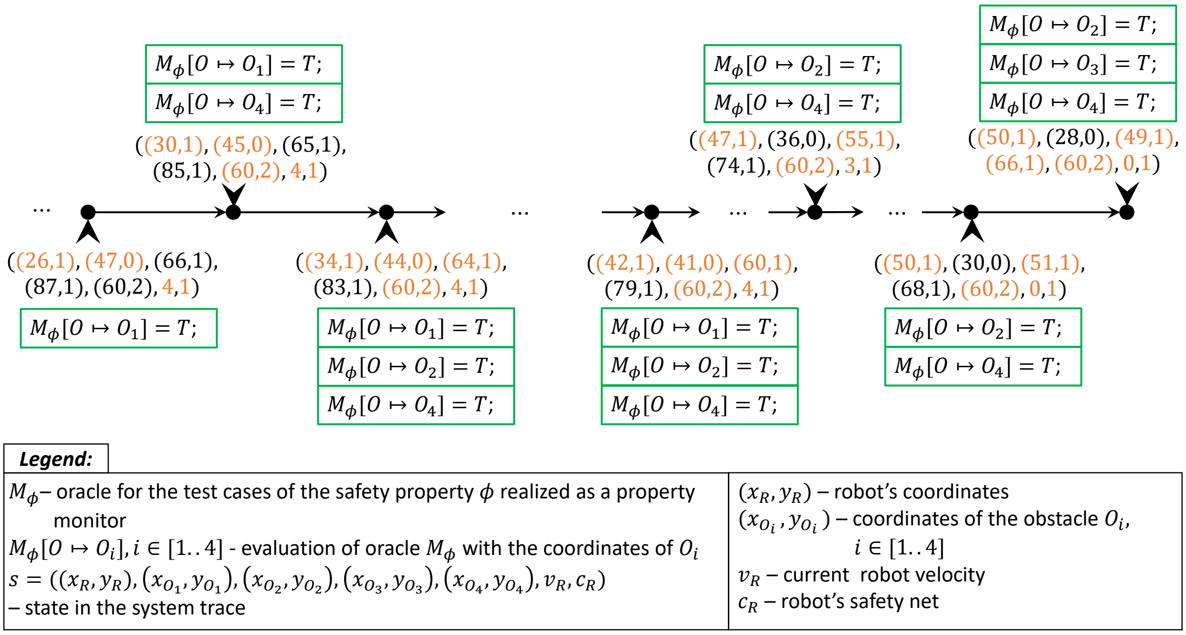

$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_4] = T;$
((30,1), (45,0), (65,1), (85,1), (60,2), 4,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((47,1), (36,0), (55,1), (74,1), (60,2), 3,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_3] = T;$
$M_\phi[O \mapsto O_4] = T;$
((50,1), (28,0), (49,1), (66,1), (60,2), 0,1)
…
…
…
…
((26,1), (47,0), (66,1), (87,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
((34,1), (44,0), (64,1), (83,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((42,1), (41,0), (60,1), (79,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((50,1), (30,0), (51,1), (68,1), (60,2), 0,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
Legend:
M_ϕ – oracle for the test cases of the safety property ϕ realized as a property monitor
$M_\phi[O \mapsto O_i], i \in [1..4]$ - evaluation of oracle M_ϕ with the coordinates of O_i
$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), (x_{O_2}, y_{O_2}), (x_{O_3}, y_{O_3}), (x_{O_4}, y_{O_4}), v_R, c_R)$ – state in the system trace
(x_R, y_R) – robot's coordinates
(x_{O_i}, y_{O_i}) – coordinates of the obstacle O_i, $i \in [1..4]$
v_R – current robot velocity
c_R – robot's safety net

Figure 3.13.: Mobile Service Robot: A System Execution and the Corresponding Evaluation of the Safety Property Monitor M_ϕ during System Test.

One relevant example is the state $((26,1),(47,0),(66,1),(87,1),(60,2),4,1)$. Notice that obstacle O_1 at coordinates $(47,0)$ is the only visible obstacle for the robot, and as such it is the only obstacle for which the evaluation of the property monitor takes place. Further down the system trace, in state $((42,1),(41,0),(60,1),(79,1),(60,2),4,1)$, the property monitor M_ϕ is evaluated not only for the obstacles O_2 and O_4, but also for the obstacle O_1. This is because the former two obstacles are situated inside the visual horizon of the robot's sensors, while the latter has driven past the robot but has not yet exited its safety net. According to Definition 3.3.3, all three are visible obstacles in the robot' sensor horizon, and as such must be monitored. The same situation occurs in the last state of the depicted system trace, in which O_2, O_3 and O_4 are the visible obstacles for the robot.

Notice the state $((50,1),(30,0),(51,1),(68,1),(60,2),4,1)$ in the system trace. In this state, the robot becomes stationary at the moment when obstacle O_2 reaches its safety net, which means that the robot's safety property is satisfied.

Table 3.5.: Mobile Service Robot: Detailed View in the Evaluation of the Safety Property Monitor M_ϕ with respect to the Visible Obstacles in the Robot's Environment.

System State	Predicate	Predicate	Predicate	Predicate	Monitor
	$p_1 : y_R == y_O$	$p_2 : x_O - x_R \geq -c_R$	$p_3 : x_O - x_R \leq c_R$	$p_4 : v_R == 0$	$M_\phi : \neg(p_1 \wedge p_2 \wedge p_3) \vee p_4$
((26,1),(47,0),(66,1),(87,1),(60,2),4,1)	$p_1[O \mapsto O_1] = F$	$p_2[O \mapsto O_1] = T$	$p_3[O \mapsto O_1] = F$	$p_4 = F$	$M_\phi[O \mapsto O_1] = T$
((30,1),(45,0),(65,1),(85,1),(60,2),4,1)	$p_1[O \mapsto O_1] = F$ $p_1[O \mapsto O_4] = F$	$p_2[O \mapsto O_1] = T$ $p_2[O \mapsto O_4] = T$	$p_3[O \mapsto O_1] = F$ $p_3[O \mapsto O_4] = F$	$p_4 = F$	$M_\phi[O \mapsto O_1] = T$ $M_\phi[O \mapsto O_4] = T$
((34,1),(44,0),(64,1),(83,1),(60,2),4,1)	$p_1[O \mapsto O_1] = F$ $p_1[O \mapsto O_2] = T$ $p_1[O \mapsto O_4] = F$	$p_2[O \mapsto O_1] = T$ $p_2[O \mapsto O_2] = T$ $p_2[O \mapsto O_4] = T$	$p_3[O \mapsto O_1] = F$ $p_3[O \mapsto O_2] = F$ $p_3[O \mapsto O_4] = F$	$p_4 = F$	$M_\phi[O \mapsto O_1] = T$ $M_\phi[O \mapsto O_2] = T$ $M_\phi[O \mapsto O_4] = T$
...	...	...	...	...	...
((42,1),(41,0),(60,1),(79,1),(60,2),4,1)	$p_1[O \mapsto O_1] = F$ $p_1[O \mapsto O_2] = T$ $p_1[O \mapsto O_4] = F$	$p_2[O \mapsto O_1] = T$ $p_2[O \mapsto O_2] = T$ $p_2[O \mapsto O_4] = T$	$p_3[O \mapsto O_1] = T$ $p_3[O \mapsto O_2] = F$ $p_3[O \mapsto O_4] = F$	$p_4 = F$	$M_\phi[O \mapsto O_1] = T$ $M_\phi[O \mapsto O_2] = T$ $M_\phi[O \mapsto O_4] = T$
...	...	...	...	...	...
((47,1),(36,0),(55,1),(74,1),(60,2),3,1)	$p_1[O \mapsto O_2] = T$ $p_1[O \mapsto O_4] = F$	$p_2[O \mapsto O_2] = T$ $p_2[O \mapsto O_4] = T$	$p_3[O \mapsto O_2] = F$ $p_3[O \mapsto O_4] = F$	$p_4 = F$	$M_\phi[O \mapsto O_2] = T$ $M_\phi[O \mapsto O_4] = T$
...	...	...	...	...	...
((50,1),(30,0),(51,1),(68,1),(60,2),0,1)	$p_1[O \mapsto O_2] = T$ $p_1[O \mapsto O_4] = F$	$p_2[O \mapsto O_2] = T$ $p_2[O \mapsto O_4] = T$	$p_3[O \mapsto O_2] = T$ $p_3[O \mapsto O_4] = F$	$p_4 = T$	$M_\phi[O \mapsto O_2] = T$ $M_\phi[O \mapsto O_4] = T$
((50,1),(28,0),(49,1),(66,1),(60,2),0,1)	$p_1[O \mapsto O_2] = T$ $p_1[O \mapsto O_3] = T$ $p_1[O \mapsto O_4] = F$	$p_2[O \mapsto O_2] = T$ $p_2[O \mapsto O_3] = T$ $p_2[O \mapsto O_4] = T$	$p_3[O \mapsto O_2] = T$ $p_3[O \mapsto O_3] = F$ $p_3[O \mapsto O_4] = F$	$p_4 = T$	$M_\phi[O \mapsto O_2] = T$ $M_\phi[O \mapsto O_3] = T$ $M_\phi[O \mapsto O_4] = T$

3.8. Requirements Validation

Requirements validation is the last stage of testing in the system development process, before the system becomes operational. In this phase, the focus lies on checking whether the developed system fulfills the customer requirements (cf. [SLS14c]). Acceptance criteria that have been agreed upon in advance with the customer are checked before the system is deployed and commercialized (cf. Section 3.2). These criteria are often described as unambiguously as possible in the contract between the system manufacturer and the customer (cf. [SLS14c]).

Along acceptance criteria agreed with the customer, further checks of the system with focus on the legal and safety regulations can take place during during requirements validation. In practice, the system manufacturer will have included the acceptance criteria as system requirements in the functional system specification as well as the relevant safety regulations in the safety analysis process, so that these can be reflected in the system design and implementation and verified during system tests. Therefore, for requirements validation, some if not all of the system tests can be rerun on on customer's site, in order to show that the defined acceptance criteria have been met (cf. [SLS14c]). Consequently, methods used for the construction of test cases for system test can be used to create test cases for requirements validation. In order to systematically derive test cases from the customer requirements, systematic methods such as automated test case generation via model checking can be applied (cf. [WRHM06], [SWRH10], [AHDR18]).

Tests performed by customers are called *acceptance tests*. These tests are carried out under realistic conditions in the physical test environment at the producer's site and in the real world at customer's site, in order to mimic the user's experience as much as possible (cf. [Mau19]). For the mobile service robot, no requirements validation has been performed. Instead, unforeseen situations which may appear in the real world on customer's site are emulated through the manipulation of the parameters of the environment model. The technical system model is subsequently checked together with the new environment model against the specified requirements (cf. Section 3.2).

3.9. Analysis of Emerging Challenges

In the previous section, design-time verification and testing have been applied on the motivational example in order to ensure the correctness of the example system with respect to the given properties specification.

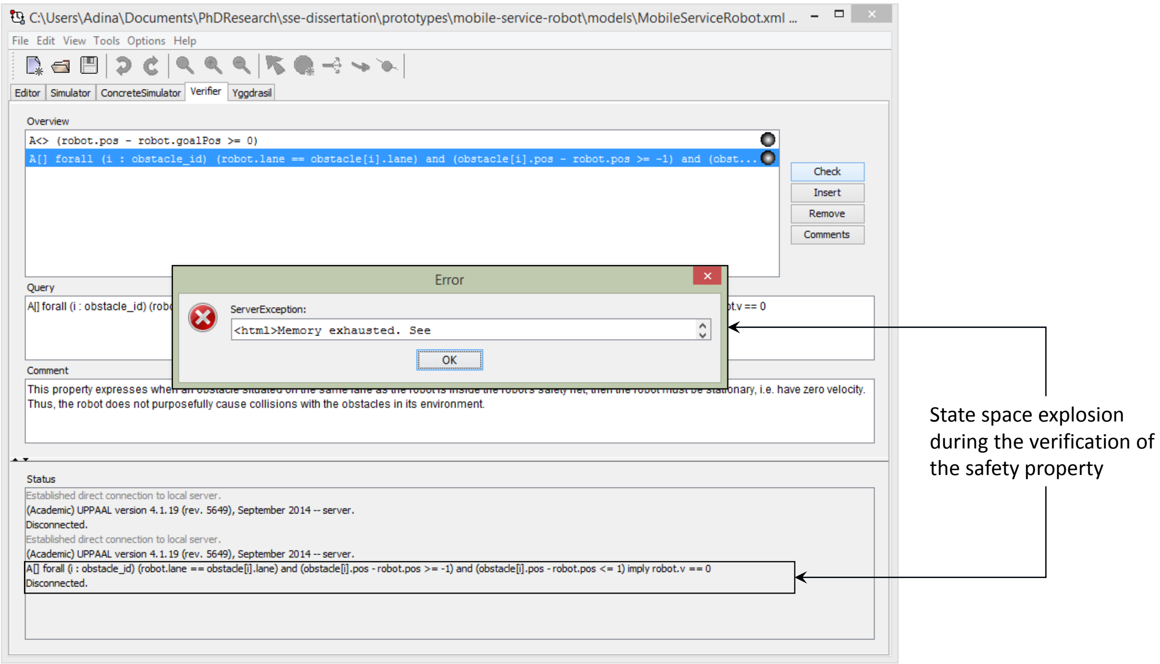
The goal of this section is to analyze the challenges raised by the application of these methods on this specific example and extrapolate these findings for the entire class of autonomous safety-critical systems, of which the example system is an instance. The issues posed by design-time verification are discussed in Section 3.9.1, while the problems raised by runtime verification are presented in Section 3.9.2.

3.9.1. Challenges of Design-Time Verification

Design-time verification methods can prove the correctness of an autonomous safety-critical system with respect to a formal system requirement specification. These methods work on the premise that any relevant information regarding the system's operational environment is known in advance at the system's design-time. This information is then used in the creation of the technical system model and of the environment model, which are given as input to the verification tool along with the formalized system requirement. Consider the example of the mobile service robot presented in Chapter 3. In this example, the information considered known a priori at the system's design-time is represented by the maximum bound for the velocities of the obstacles in the robot's environment. The value of the maximum bound is then used to model the computations performed during the planing phase, i.e., collision distance calculations. In Section 3.5, the maximum bound for the obstacles velocities is considered to be $v^{O}_{Max} = 2$. The three dynamic obstacles present in the robot's environment have their specific maximum velocities set at $v^{O_i}_{Max} = 2, i \in \{1, 2, 3\}$. In these conditions, Figure 3.12 shows the expected result for the verification of the robot's safety property, i.e., the property is satisfied.

Autonomous safety-critical systems are highly complex systems which receive heterogeneous sensor data as input from their environment. This raises scalability issues with design-time verification methods. Specifically, model checking techniques suffer from the state space explosion problem which limits drastically the size of the systems that can be verified. Furthermore, unforeseen changes may occur in the system's operational environment after the release and commissioning of the autonomous system. By definition, design-time verification methods applied to autonomous systems can reason only on the basis of information available at design-time. Although the technical system model and the environment model can be extended in further iterations of the development process in order to address the changes appeared in the operational environment, this can lead to an overall system model which may be too large for the design-time verification to give a result in a reasonable time. In order to illustrate this, consider the initial state of the overall system model represented in Figure 3.10 to which the system designer brings the following changes. The environment model is modified so that the dynamic obstacles in the robot's environment move with a specific maximum velocity of $v^{O_i}_{Max} = 3, i \in \{1, 2, 3\}$. In response to the changes in the environment model, the technical system model is adjusted accordingly by setting the maximum bound for the obstacles velocities also to $v^{O}_{Max} = 3$. Verifying the robot's safety property in these conditions should yield the same result as before, i.e., the safety property should be satisfied. However, the modification of the specific maximum velocities of the dynamic obstacles leads to an exponential increase in the size of the overall system model, and as a consequence, to the state space explosion problem. Figure 3.14 illustrates a view of the verifier panel of the Uppaal model checker, in which the verification of the robot's safety property does not terminate due to the exhaustion of the memory space.

Moreover, there is an uncountable number of unforeseeable situations that can occur in the real environment, which makes it impossible to construct a complete model of the environment at design-time.



C:\Users\Adina\Documents\PhDResearch\sse-dissertation\prototypes\mobile-service-robot\models\MobileServiceRobot.xml ...

Overview

```
A<> (robot.pos - robot.goalPos >= 0)
A[] forall (i : obstacle_id) (robot.lane == obstacle[i].lane) and (obstacle[i].pos - robot.pos >= -1) and (obst...
```

Query

A[] forall (i : obstacle_id) (rob… …t.v == 0

ServerException:

```
<html>Memory exhausted. See
```

Comment

This property expresses when … …onary, i.e. have zero velocity. Thus, the robot does not purposefully cause collisions with the obstacles in its environment.

Status

```
Established direct connection to local server.
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.1.19 (rev. 5649), September 2014 -- server.
A[] forall (i : obstacle_id) (robot.lane == obstacle[i].lane) and (obstacle[i].pos - robot.pos >= -1) and (obstacle[i].pos - robot.pos <= 1) imply robot.v == 0
Disconnected.
```

State space explosion during the verification of the safety property

Figure 3.14.: Mobile Service Robot: State Space Explosion Problem Visualized in UPPAAL.

3.9.2. Challenges of Testing

System test can be used complementary to design-time verification in order to overcome the presented scalability issues of the latter and add more flexibility in checking that the behavior of the system under test fulfills the system requirements specification. For specific application domains, e.g., the automotive industry, testing is considered to be the main method of verification (cf. [Int11f], [Int11a]). Test oracles can be used to detect the incorrect behavior of an autonomous system with respect to a given property specification when the system under test executes a defined test case. The test oracle can be extracted from the property specification itself in the form of a property monitor. The monitor observes an execution of the system and the result of the system execution is compared with the expected result in each observed system state. If there exists a state in the system execution for which the result computed by the system under test deviates from the expected result, then the property monitor returns the verdict *false*, signaling that the property specification is falsified by the respective test case.

Testing as a method to verify the behavior of the robot in its physical environment presents several challenges. Consider that the system developers want to test the following scenario. After the release of the mobile service robot, the physical environment for which the robot was planned may evolve and not correspond anymore to the environment model developed at design-time. To test such a situation, the test engineers deploy the system in a controlled test environment, in which they have introduced several changes with respect to the environment model developed during the system's design phase. These changes represent the increased specific maximum velocities of the obstacles O_1 and O_3, namely $v_{Max}^{O_1} = v_{Max}^{O_3} = 3$. At the same time the test engineers leave unchanged the maximum bound for the obstacles velocities used by the robot in the computation of its collision distance, i.e., $v_{Max}^{O} = 2$. This is to reflect that the system designers are not aware of the changes occurred in the controlled test environment with respect to $v_{Max}^{O_1}$ and $v_{Max}^{O_3}$. The initial system state effective at the robot's system test in the controlled test environment is depicted in Figure 3.15.

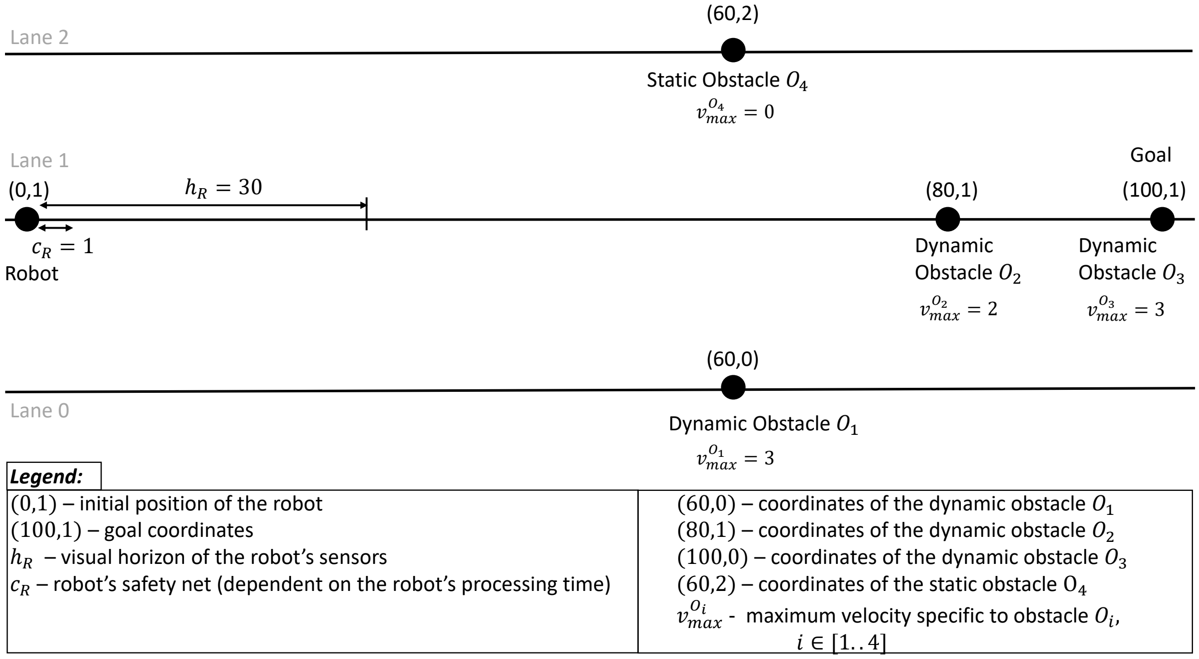

(60,2)
Lane 2
Static Obstacle O_4
$v_{max}^{O_4} = 0$
Lane 1
Goal
(0,1)
$h_R = 30$
(80,1)
(100,1)
$c_R = 1$
Robot
Dynamic Obstacle O_2
$v_{max}^{O_2} = 2$
Dynamic Obstacle O_3
$v_{max}^{O_3} = 3$
(60,0)
Lane 0
Dynamic Obstacle O_1
$v_{max}^{O_1} = 3$
Legend:
(0,1) – initial position of the robot
(100,1) – goal coordinates
h_R – visual horizon of the robot's sensors
c_R – robot's safety net (dependent on the robot's processing time)
(60,0) – coordinates of the dynamic obstacle O_1
(80,1) – coordinates of the dynamic obstacle O_2
(100,0) – coordinates of the dynamic obstacle O_3
(60,2) – coordinates of the static obstacle O_4
$v_{max}^{O_i}$ - maximum velocity specific to obstacle O_i, $i \in [1..4]$

Figure 3.15.: Mobile Service Robot: A Visual Representation of the Initial System State during System Test.

Figure 3.16 depicts a system execution of the mobile service robot along with the corresponding trace of the property monitor M_ϕ in the context of the newly changed physical environment. Notice the system state $((49,1),(32,0),(57,1),(70,1),(60,2),3,1)$ in which the robot starts to brake due to the collision danger represented by obstacle O_2. In this state, both obstacles O_2 and O_4 are visible to the robot's sensors and monitored by the property monitor M_ϕ. Even though the obstacle O_3 has a specific maximum velocity larger than that of O_2, the former is not necessarily faster than the latter, as shown in the system execution depicted in Figure 3.16. This proves to be an advantage for the robot. The robot's brake maneuver ends in the system state $((52,1),(23,0),(53,1),(61,1),(60,2),0,1)$, as the robot comes to a standstill and waits for obstacle O_2 to pass by. Once the obstacle O_2 has left its safety net, the robot may resume its drive towards its destination. On its way, the robot detects another collision danger represented by obstacle O_3. At this moment, the robot has several alternatives: brake to full stop and wait for the obstacle to drive by or change to a safe lane.

Notice that from the point of view of the system test, the situation presented in Figure 3.16 is identical to the one illustrated in Figure 3.13. This is because the property monitor cannot detect the evolution which has taken place in the physical environment of the robot during the system test, i.e., the specific maximum velocities of obstacles O_1 and O_3 are larger than the maximum bound used by the system designer at design-time.

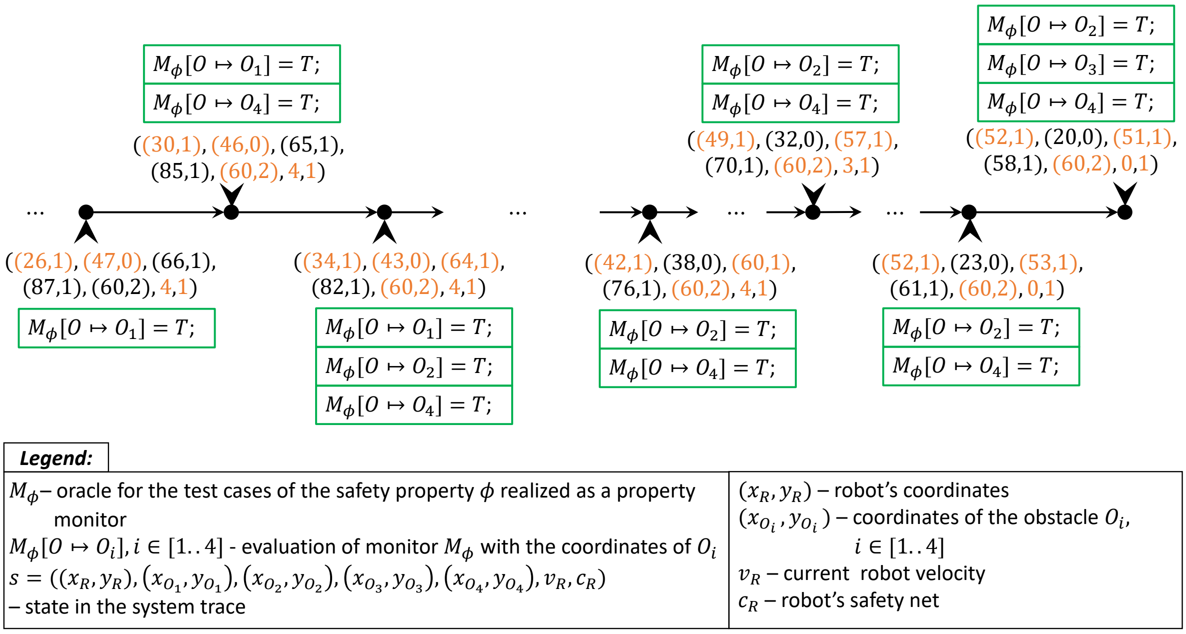

$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_4] = T;$
((30,1), (46,0), (65,1), (85,1), (60,2), 4,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((49,1), (32,0), (57,1), (70,1), (60,2), 3,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_3] = T;$
$M_\phi[O \mapsto O_4] = T;$
((52,1), (20,0), (51,1), (58,1), (60,2), 0,1)
...
...
...
...
((26,1), (47,0), (66,1), (87,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
((34,1), (43,0), (64,1), (82,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((42,1), (38,0), (60,1), (76,1), (60,2), 4,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((52,1), (23,0), (53,1), (61,1), (60,2), 0,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
Legend:
M_ϕ– oracle for the test cases of the safety property ϕ realized as a property monitor
$M_\phi[O \mapsto O_i], i \in [1..4]$ - evaluation of monitor M_ϕ with the coordinates of O_i
$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), (x_{O_2}, y_{O_2}), (x_{O_3}, y_{O_3}), (x_{O_4}, y_{O_4}), v_R, c_R)$
– state in the system trace
(x_R, y_R) – robot's coordinates
(x_{O_i}, y_{O_i}) – coordinates of the obstacle O_i, $i \in [1..4]$
v_R – current robot velocity
c_R – robot's safety net

Figure 3.16.: Mobile Service Robot: Evaluation of the Safety Property Monitor M_ϕ showing the Safety Property ϕ fulfilled during System Test.

Nevertheless, the evolution occurred in the physical test environment of the robot can lead to situations in which the robot actively collides with obstacles in its environment, thus violating its safety property. To illustrate this, consider the system execution depicted in Figure 3.17 which takes place under the same initial conditions. Notice the system state $((44,1),(24,0),(67,1),(64,1),(60,2),4,1)$ in which obstacle O_2 is overtaken by obstacle O_3, as the latter moves with maximum velocity and therefore is faster than the former. Since the maximum bound used for obstacles velocities at design-time is smaller than the actual velocity of O_3, the monitor M_ϕ detects a violation of the robot's safety property in the system state $((55,1),(17,0),(61,1),(56,1),(60,2),3,1)$, i.e., the robot is still moving at the moment when the dynamic obstacle O_3 has reached its safety net.

In contrast to design-time verification, which looks at the entire system state space, system test uses property monitors to observe only the system states during the current system execution. Therefore testing is incomplete and can only show the presence of errors in a system, but is not adequate to show their absence (cf. [Dij72]). This is illustrated in Figure 3.17, in which a violation of the robot's safety property occurs, i.e., the robot actively collides with obstacle O_3. However, in order to ensure its safe operation in its operational environment, an autonomous safety-critical system must be thoroughly tested. In the automotive domain, to be certified as safe with fulfillment of its designated ASIL, a safety-critical automotive system must pass specific tests, e.g.,

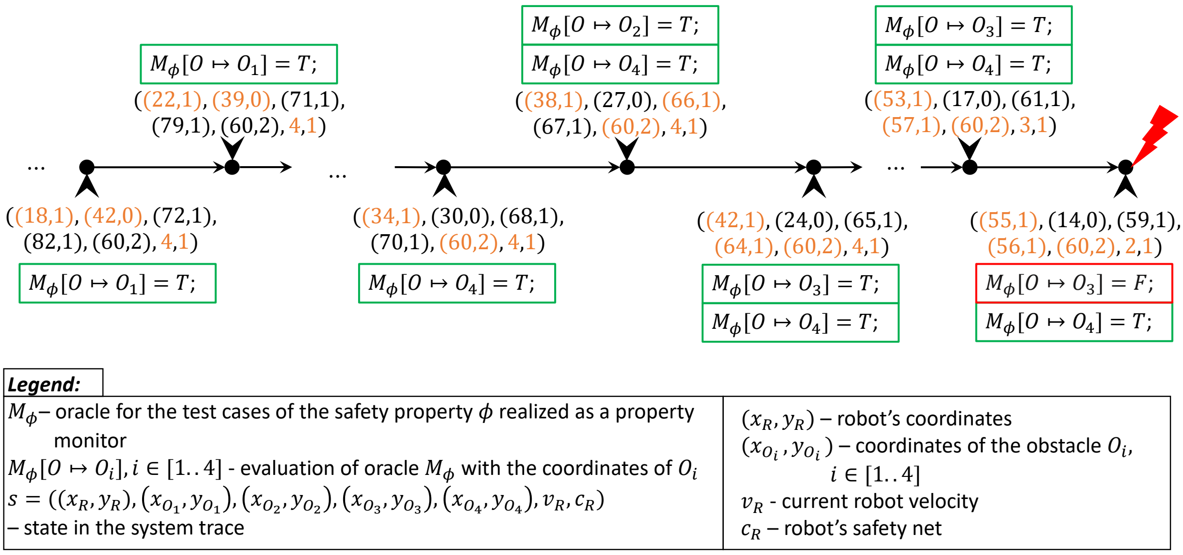

$M_\phi[O \mapsto O_1] = T;$
((22,1), (39,0), (71,1), (79,1), (60,2), 4,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
((38,1), (27,0), (66,1), (67,1), (60,2), 4,1)
$M_\phi[O \mapsto O_3] = T;$
$M_\phi[O \mapsto O_4] = T;$
((53,1), (17,0), (61,1), (57,1), (60,2), 3,1)
...
...
...
((18,1), (42,0), (72,1), (82,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
((34,1), (30,0), (68,1), (70,1), (60,2), 4,1)
$M_\phi[O \mapsto O_4] = T;$
((42,1), (24,0), (65,1), (64,1), (60,2), 4,1)
$M_\phi[O \mapsto O_3] = T;$
$M_\phi[O \mapsto O_4] = T;$
((55,1), (14,0), (59,1), (56,1), (60,2), 2,1)
$M_\phi[O \mapsto O_3] = F;$
$M_\phi[O \mapsto O_4] = T;$
Legend:
M_ϕ – oracle for the test cases of the safety property ϕ realized as a property monitor
$M_\phi[O \mapsto O_i], i \in [1..4]$ - evaluation of oracle M_ϕ with the coordinates of O_i
$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), (x_{O_2}, y_{O_2}), (x_{O_3}, y_{O_3}), (x_{O_4}, y_{O_4}), v_R, c_R)$ – state in the system trace
(x_R, y_R) – robot's coordinates
(x_{O_i}, y_{O_i}) – coordinates of the obstacle O_i, $i \in [1..4]$
v_R - current robot velocity
c_R – robot's safety net

Figure 3.17.: Mobile Service Robot: Evalution of the Safety Property Monitor M_ϕ showing the Safety Property ϕ disproved during System Test.

road tests required by ASIL C/D. Yet, it is infeasible to test with the required level of thoroughness, e.g., validation of a $\frac{1}{10^9}$ h for an autonomous vehicle would require billions of hours of driving in representative environments. In addition, test may have to be repeated several times in order to achieve statistical significance (cf. [KW16]).

Moreover, testing cannot fully address controllability challenges of the autonomous system. Notice that the system is the primary exception handler in case of beyond-specified operational conditions as illustrated in Figure 3.17. The inherent uncertainty in the operational environment introduces the issue of nondeterminism in testing. Consequently, certain edge cases that may be of interest to the system designer are difficult to exercise, as it is difficult to construct a situation in which the physical environment provides the necessary conditions to run a particular test case (cf. [KW16]).

3.10. Scope of this Thesis

Based on the analysis performed in the previous section, it is this section's purpose to define the scope of this work. Thus, the section starts with an introduction of Runtime Monitoring of Environment Assumptions, the novel approach proposed by this work (Section 3.10.1). It continues by analyzing the challenges that arise from this approach and outlines the scope of this work by defining four research questions that are to be addressed in this thesis (Section 3.10.2).

3.10.1. Introduction of Runtime Monitoring of Environment Assumptions

In the motivational example, model checking is performed under the premise that the dynamic obstacles in the robot's environment move with a maximum velocity known beforehand by the system designer. This premise is not formulated as an explicit assumption, yet it is used in the algorithm for the collission distance computation performed by the mobile service robot. The collision distance computation is part of the technical system model, which put together with the environment model through parallel composition in an overall system model. The overall system model and a formalized system requirement are given as inputs to the model checking tool. If the verification terminates, then the model checker gives an answer to the question whether the overall system model satisfies the formalized system requirement or not. In case of large models, design-time verification methods show scalability issues. Specifically, model checking has been shown to suffer from the state space explosion problem, which limits considerably the size of the models that can be verified.

System test is used to complement design-time verification and overcome its scalability issues. During system test, monitors extracted from the formal specification of the robot's requirements used to check whether the behavior of the robot fulfills the system requirements. During testing, the robot is stimulated with test input data and the monitor observes its execution, comparing the result computed by the robot with the result expected by the monitor in each observed system state. The monitor returns the verdict *false* in case there is a state in which the result computed by the system deviates from the expected result. However, after the robot's release and commissioning in its operational environment, the implicit assumptions used during the system design may not be valid. This is possible either because these assumptions were false from the start or the environment for which the robot has been planned has evolved in such a way that the premises under which the robot has been designed and developed are no longer valid. Furthermore, the monitors extracted from the formal requirements specification cannot detect whether these assumptions are valid or not, because there is not explicit definition of these assumptions design-time. Invalid assumptions can lead to situations in which the robot actively collides with an obstacle, thus violating its safety property.

In order to address this issue, this thesis proposes a novel approach which extends the proof objectives to checking whether the premises under which the system has been developed remain valid at runtime. Specifically, for the purpose of this approach, the concept of environment assumptions is defined.

Definition 3.10.1 - Environment Assumption
An environment assumption represents a characteristic of the real environment of an autonomous safety-critical system. This characteristic is presumed known at the design-time of the system. Reasoning about the satisfiability of an environment assumption is performed on the basis of the system's observations of its real environment. An example of such an assumption is the maximum bound for the obstacles velocities used by the robot in the motivational example for the computations of its collision distance. ■

During design and development of autonomous safety-critical systems, environment assumptions are used in designing the algorithms which help control such systems. In order to reason about the correctness of such systems with respect to a given property specification, this thesis proposes an approach complementary to design-time verification, which explicitly describes the environment assumptions and uses this description as a basis for building the runtime monitors. This approach is referred to as Runtime Monitoring of Environment Assumptions.

3.10.2. Research Questions of this Work

The previous section introduced the reader to the general idea of the approach Runtime Monitoring of Environment Assumptions, while the purpose of this section is to discuss the challenges brought forth by this approach and outline the research questions of this thesis. These challenges must be overcome in order for the approach presented in this work to be applicable.

Based on the problem analysis performed in this chapter, the general research question of this thesis is formulated as follows:

> *How can design-time guaranteed environment assumptions be used at runtime to continuously validate the design-time verification result with respect to autonomous safety-critical systems operating in uncertain environments?*

The first challenge of the proposed approach is to find an adequate form to express in the most general way possible the environment assumptions of a autonomous safety-critical system. For this purpose, a description language is needed to describe assumptions in the from of quantities such maximum obstacle velocity, but also dynamic aspects such as the behavior of dynamic obstacles over the course of time. Besides being expressive enough, the description language for environment assumptions must be kept as simple as possible in order for it to be amenable to runtime monitoring. Thus, based on the general question, it is possible to derive the first research question of this work:

Research Question 1 (RQ-1) How can environment assumptions and their relation to the system's safety requirement specification be explicitly and formally described?

In order to answer RQ-1, a method anchored in the phases of the system development process will be used to explicitly define the environment assumptions at system's design-time. Firstly, a requirements analysis will be carried out on a revisited version of the motivational example. The next step will be a safety analysis using as input the description of the revisited motivational example and the system requirements elicited through the requirements analysis. The result of safety analysis will be used to produce an informal specification of the environment assumptions, which are seen as an extension of the informal specification of the system's safety requirements obtained during the phase of requirements analysis. Using a requirement pattern designed explicitly for this purpose, the safety requirements specification will be systematically extended with the specification of the environment assumptions. The result is a catalog of extended safety requirements, which will be formalized with the help of an appropriate formal language.

Thus, the answer to the first research question is a method for the explicit definition and formalization of environment assumptions using the phases of the system development process. This method will not only formally describe the environment assumptions of an automated safety-critical system, but also emphasizes their relationship to its safety requirement specification.

Reasoning about a system's correctness at runtime will be done not by monitoring the safety requirement specification of the system. Instead, its environment assumptions are subject to observation and monitoring. For this purpose, the formal description of the environment assumptions developed in response to RQ-1 is used to construct the corresponding runtime monitors. These monitors are henceforth denoted as environment assumptions monitors. Thus, the second research question of this thesis can be defined as follows:

Research Question 2 (RQ-2) How can the formal description in RQ-1 be used to construct environment assumptions monitors?

In order to answer RQ-2, a method is defined for the construction of runtime monitors using the formal description of the environment assumptions.

The approach presented in this thesis must be applied on several real-world example systems in order to demonstrate its applicability. Therefore, a third and last research question is derived from the general question:

Research Question 3 (RQ-3) How can the applicability of this approach be demonstrated for real-world safety-critical systems?

In order to answer RQ-3, two case studies will be built around two real-world autonomous safety-critical systems, a mobile service robot and automotive system function used for the speed estimation of a moving vehicle. The RMEA concept will be applied for each of these systems, explicitly defining environment assumptions at system's design time and using runtime monitoring to validate them during system's operation.

The final purpose is to investigate the extent to which the runtime monitors of the environment assumptions can guarantee the system correctness with respect to its property specifications. Thus, it will be demonstrated that Runtime Monitoring of Environment Assumptions is a suitable approach for extending to operation time the guarantees obtained at design-time for the correctness of automated safety-critical systems.

3.11. Summary

The goal of this chapter was to perform a problem analysis and clearly highlight the issues which will be addressed in this thesis. In order to do this, the chapter started with the introduction of a motivational example. Current state of the art verification methods have been applied on the example and an analysis of the ensuing challenges has been carried out. Consequently, the approach Runtime Monitoring of Environment Assumptions has been introduced and the challenges that emerge from it have been analyzed.

On the basis of this analysis, the scope of this work has been outlined through the definition of the following general research question:

> *How can design-time guaranteed environment assumptions be used at runtime to continuously validate the design-time verification result with respect to autonomous safety-critical systems operating in uncertain environments?*

Further, the following three research questions have been derived from the general research question:

Research Question 1 (RQ-1) How can environment assumptions and their relation to the system's property specification be explicitly and formally described?

Research Question 2 (RQ-2) How can the formal description in RQ-1 be used to construct environment assumptions monitors?

Research Question 3 (RQ-3) How can the applicability of this approach be demonstrated for real-world case studies?

Chapter 4.

Solution Concept

4.1. Runtime Monitoring of Environment Assumptions 130
4.1.1. Integration in the System Development Process 131
4.1.2. Overview of Concept 132
4.1.3. Runtime Monitoring of Environment Assumptions by Example . . 136
4.2. Revisiting the Motivational Example 143
4.3. Requirements Elicitation and Analysis 149
4.3.1. Informal Specification of System Requirements 149
4.3.2. Formal Specification of System Requirements 154
4.4. Safety Analysis 156
4.4.1. HARA Analysis of the Revisited Motivational Example 156
4.4.2. “Safe Enough” for Autonomous Safety-Critical Systems 161
4.4.3. Extending Safety Requirements with Environment Assumptions . 163
4.4.4. Informal Specification of Extended Safety Requirements 165
4.4.5. Formal Specification of Extended Safety Requirements 167
4.5. System Design 174
4.5.1. Environment Model 174
4.5.2. Technical System Model 180
4.5.3. Design Time Verification 185
4.5.4. Analysis of the Environment Assumptions 186
4.5.5. Formal Definition of Environment Assumptions Monitors 188
4.6. System Implementation 191
4.6.1. Implementation of the System Model 191
4.6.2. Realization of the Environment Model 192
4.6.3. Realization of the Environment Assumptions Monitors 192
4.7. System Test 193
4.7.1. Testing the Implemented System 194
4.7.2. Testing Environment Assumptions Monitors 194
4.8. Requirements Validation 194
4.8.1. Environment Assumptions Validation via Runtime Monitoring . . 194
4.9. Summary 195

In Section 3.10.2 the research questions which are to be tackled by this work were introduced and motivated. Research question RQ-1 deals with finding an appropriate method for the explicit and formal definition of environment assumptions, while the research question RQ-2 aims to use this formal definition for the construction of environment assumptions monitors. The goal of this chapter is to address these two research questions and to describe in detail the approaches that will be used to solve the two research questions. In order to achieve this goal, this chapter produces the following two artifacts as output:

Method for Explicit Definition of Environment Assumptions. The first artifact is a method for the explicit definition of environment assumptions at th design time of automated safety-critical systems using an appropriate formal language. This method defines not only the environment assumptions but also emphasizes their relation to the safety requirement specification of the system. This artifact contributes directly to research question RQ-1.

Method for Construction of Environment Assumptions Monitors. The second artifact of this chapter is a method to build monitors for the environment assumptions based on their explicit formal definition. This artifact contributes directly to research question RQ-2.

In order to create these artifacts, the chapter is structured as follows. Section 4.1 discusses the integration of the runtime monitoring of the environment assumptions in the development process from Chapter 3 and gives an overview of the concept on the basis of a simple example. Section 4.2 revisits the motivational example and introduces further environmental factors, which are relevant for the specification of environment assumptions in the given example. The following sections go through each phase of the development process and emphasize the additional steps necessary for the integration of the runtime monitoring of environment assumptions, starting with the requirements analysis (Section 4.3) and safety analysis (Section 4.4), continuing with system design and implementation (Section 4.5 and Section 4.6), and ending with system test (Section 4.7) and requirements validation (Section 4.8). Section 4.4 and Section 4.5 are strongly related to research questions RQ-1 and RQ-2 respectively, as these sections create the two artifacts of this chapter, the environment assumptions description language and the monitor construction method. Section 4.9 concludes with a summary of the ideas and concepts presented in this chapter.

4.1. Runtime Monitoring of Environment Assumptions

As seen in Chapter 3, environment assumptions are an integral part of the system design process and may exist and may be used implicitly at design-time. However, in order to verify autonomous safety-critical systems, an explicit definition of the environment assumptions specification is necessary at design-time. This thesis proposes a concept by which environment assumptions can be defined explicitly and formally specified at design-time. The formal specification of environment assumptions is then used as a basis for the construction of their respective monitors. The monitors are, in turn, used for the validation of the environment assumptions during the system's operation. Section 4.1.1 illustrates the integration of this concept in the overall system development process, while Section 4.1.2 gives an overview of the concept and explains on the basis of a simple example its general functioning.

4.1.1. Integration in the System Development Process

For the development of the example system, the V-model is the chosen development process, since its plan-driven approach is appropriate for safety-critical systems. In order to properly introduce the approach of Runtime Monitoring of Environment Assumptions (RMEA), activities existent in the current development process must be extended accordingly. Figure 4.1 depicts the necessary extensions undertaken in the system development process.

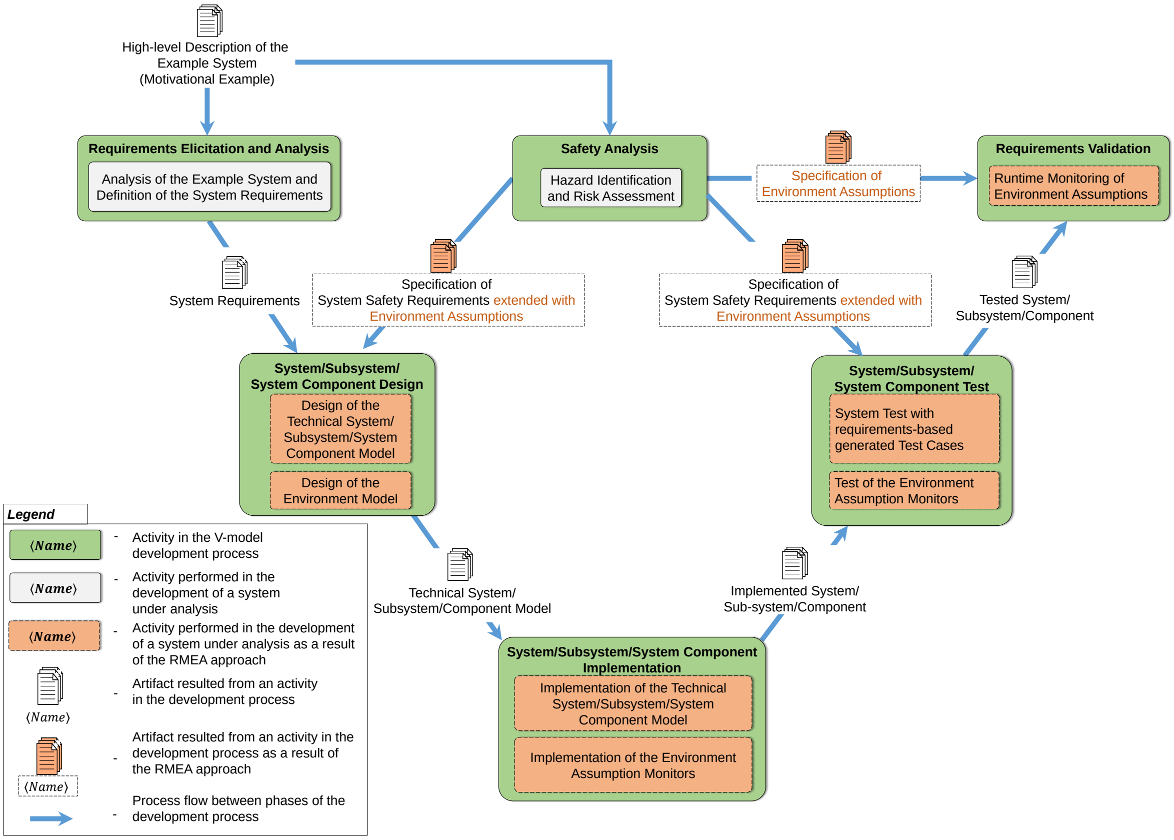

High-level Description of the Example System (Motivational Example)
Requirements Elicitation and Analysis
Analysis of the Example System and Definition of the System Requirements
Safety Analysis
Hazard Identification and Risk Assessment
Specification of Environment Assumptions
Requirements Validation
Runtime Monitoring of Environment Assumptions
System Requirements
Specification of System Safety Requirements extended with Environment Assumptions
Specification of System Safety Requirements extended with Environment Assumptions
Tested System/ Subsystem/Component
System/Subsystem/ System Component Design
Design of the Technical System/ Subsystem/System Component Model
Design of the Environment Model
System/Subsystem/ System Component Test
System Test with requirements-based generated Test Cases
Test of the Environment Assumption Monitors
Technical System/ Subsystem/Component Model
Implemented System/ Sub-system/Component
System/Subsystem/System Component Implementation
Implementation of the Technical System/Subsystem/System Component Model
Implementation of the Environment Assumption Monitors
Legend
⟨Name⟩ - Activity in the V-model development process
⟨Name⟩ - Activity performed in the development of a system under analysis
⟨Name⟩ - Activity performed in the development of a system under analysis as a result of the RMEA approach
⟨Name⟩ - Artifact resulted from an activity in the development process
⟨Name⟩ - Artifact resulted from an activity in the development process as a result of the RMEA approach
- Process flow between phases of the development process

Figure 4.1.: Overall System Development Process: the changes pertaining to this thesis' approach are highlighted in orange.

The phase of safety analysis is the first phase of the development process in which specific activities in support of the RMEA approach are undertaken. In this phase, hazard identification and risk analysis is used in order to identify hazards which may occur due to invalid environment assumptions and derive an appropriate specification of the system safety requirements. The artifact resulting from the safety analysis phase is the system safety requirements specification extended with a specification of the environment assumptions. The extended safety requirements specification is used along with the system requirements resulted from the requirements elicitation and analysis to carry out the system design phase. As the system design progresses, the system is refined down to system component level. For the various levels of abstraction in the system design, the system designers create the corresponding artifacts, e.g., technical models which describe the structure and the behavior of the system components, of the system's sub-systems, and of the system itself. In addition to the technical system model, the system design phase specifies also a model of the environment in which the system is designed to operate. These models serve then as a basis for the implementation of the respective part of the system. Along the implementation of the technical system model, the system engineers design and implement specific monitoring components, which will be used to check the validity of the environment assumptions during the system's operation. These components are henceforth denoted as *environment assumptions monitors*. During the system test phase, the implemented system as well as the environment assumptions monitors are tested. The implemented system is tested using requirements-based test cases obtained automatically via model checking. In turn, the test cases for testing of the environment assumption monitors are built so that they reflect violations of the environment assumptions and check whether these violations trigger the respective environment assumption monitors. The system test phase is followed by the requirements validation phase, in which the environment assumptions of the system under analysis are validated in the system's operational environment.

Notice that the RMEA approach is applicable at any system hierarchy level. The environment of the test object may vary depending on the hierarchy level at which the test object is situated. If the test object is represented by the whole system to be deployed, then the corresponding environment is the physical world. However, if the test object is a system component or a subsystem of the system under analysis, then the corresponding environment is a combination of a technical environment and the physical world. At component and subsystem level, the technical environment of the test object consists of the hardware and software components and subsystems with which the test object communicates through its interfaces. During the test phase, the test object is run in parallel to its environment and is stimulated with inputs from it. During the phase of requirements validation, the environment assumptions are validated during the system's execution in its operational environment.

4.1.2. Overview of Concept

The problem analysis performed in Chapter 3 has shown on the basis of an example that, with respect to the detection of property violations in autonomous safety-critical systems, both design-time verification and system testing raise several issues. In the motivational example, design-time verification uses the premise that maximum velocity with which the dynamic obstacles move in the robot's environment is known in advance at system design-time. This premise is not formulated as an explicit assumption, yet it is used in the design of the collision distance algorithm performed by the mobile service robot. The collision distance computation is part of the technical system model, which is put together with the environment model via parallel composition in an overall system model. The latter is then verified against a formalized system requirement via model checking. If the verification terminates, then the model checker gives an answer to the question whether the overall system model satisfies the formalized system requirement or not. In case the overall system model does not satisfy the formalized system requirement, then the model checker returns a system trace as a counterexample. The counterexample is usually represented as a sequence of system states starting in the initial state of the overall system model and ending with the error state. In case of large models, design-time verification methods show scalability issues. Specifically, model checking has been shown to suffer from the state space explosion problem, which limits considerably the size of the models that can be verified.

System test is used to complement design-time verification and overcome its scalability issues. During system test, monitors derived from the formal specification of the robot's requirements are used to check whether the robot's behavior fulfills the system requirements. Throughout the system test, the robot is stimulated with test input data and the monitor observes its execution, comparing the result computed by the robot with the result expected by the monitor in each observed system state. The monitor returns the verdict *false* in case there is a state in which the result computed by the system deviates from the expected result. However, after the robot's release and commissioning in its operational environment, the implicit assumptions used during the system design may not be valid. Furthermore, the monitors extracted from the formal requirements specification cannot detect whether these assumptions are valid or not, because there is not explicit definition of these assumptions at design-time. Invalid assumptions can lead to situations in which the robot actively collides with an obstacle, thus violating its safety property.

In order to address these limitations, this thesis proposes RMEA as a novel approach to extend the quality assurance goals of autonomous safety-critical systems towards the runtime monitoring and validation of environment assumptions explicitly defined at design-time.

RMEA is a safety engineering approach for autonomous safety-critical systems, which is integrated in the system development process of such systems (see Figure 4.1). An overview of this approach is depicted in Figure 4.2. The approach is anchored in the system development process and uses the artifacts produced in the different phases of the development process. There are three phases of the development process in which the RMEA approach plays a decisive role: system design, system test, and requirements validation. Each of these phases defines specific *quality assurance goals* that contribute to the realization of the RMEA approach and are highlighted in Figure 4.2.

During *system design*, two abstract models, *SM* and *EM*, are constructed in order to describe the desired system's behavior and the a priori knowledge of its operational environment, which the system designer has at design-time. In accordance with the definitions introduced in Section 2.5.2, *SM* is the technical system model, while *EM* represents the environment model. The quality assurance goal of the system design

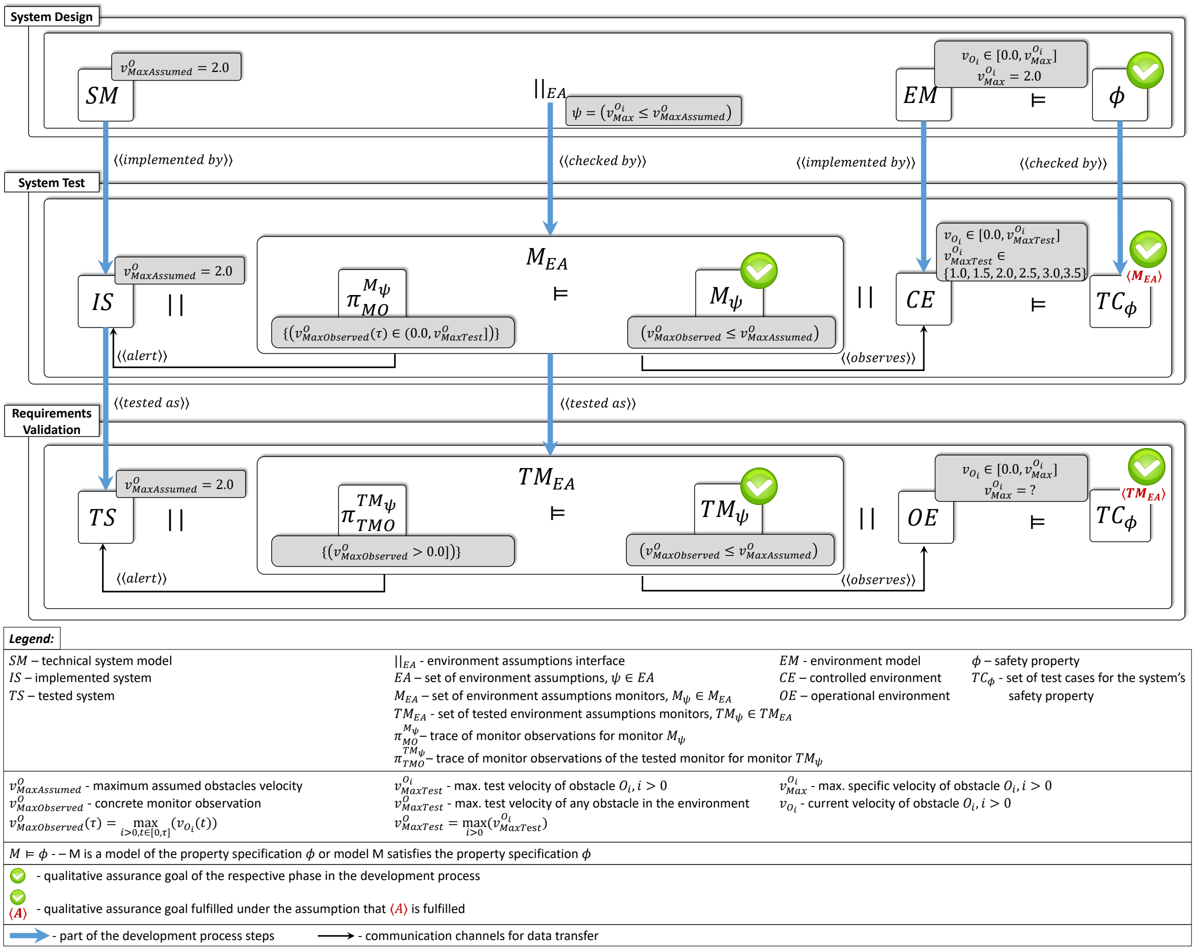

System Design
$v^{O}_{MaxAssumed} = 2.0$
SM
$||_{EA}$
$\psi = (v^{O_i}_{Max} \leq v^{O}_{MaxAssumed})$
$v_{O_i} \in [0.0, v^{O_i}_{Max}]$
$v^{O_i}_{Max} = 2.0$
EM
$\models$
ϕ
⟨⟨implemented by⟩⟩
⟨⟨checked by⟩⟩
⟨⟨implemented by⟩⟩
⟨⟨checked by⟩⟩
System Test
$v^{O}_{MaxAssumed} = 2.0$
IS
$||$
M_{EA}
$\models$
$\pi^{M_\psi}_{MO}$
$\{(v^{O}_{MaxObserved}(\tau) \in (0.0, v^{O}_{MaxTest}])\}$
M_ψ
$(v^{O}_{MaxObserved} \leq v^{O}_{MaxAssumed})$
$||$
$v_{O_i} \in [0.0, v^{O_i}_{MaxTest}]$
$v^{O_i}_{MaxTest} \in \{1.0, 1.5, 2.0, 2.5, 3.0, 3.5\}$
CE
$\models$
$\langle M_{EA} \rangle$
TC_ϕ
⟨⟨alert⟩⟩
⟨⟨observes⟩⟩
⟨⟨tested as⟩⟩
⟨⟨tested as⟩⟩
Requirements Validation
$v^{O}_{MaxAssumed} = 2.0$
TS
$||$
TM_{EA}
$\models$
$\pi^{TM_\psi}_{TMO}$
$\{(v^{O}_{MaxObserved} > 0.0])\}$
TM_ψ
$(v^{O}_{MaxObserved} \leq v^{O}_{MaxAssumed})$
$||$
$v_{O_i} \in [0.0, v^{O_i}_{Max}]$
$v^{O_i}_{Max} = ?$
OE
$\models$
$\langle TM_{EA} \rangle$
TC_ϕ
⟨⟨alert⟩⟩
⟨⟨observes⟩⟩
Legend:
SM – technical system model
IS – implemented system
TS – tested system
$||_{EA}$ - environment assumptions interface
EA – set of environment assumptions, $\psi \in EA$
M_{EA} – set of environment assumptions monitors, $M_\psi \in M_{EA}$
TM_{EA} - set of tested environment assumptions monitors, $TM_\psi \in TM_{EA}$
$\pi^{M_\psi}_{MO}$ – trace of monitor observations for monitor M_ψ
$\pi^{TM_\psi}_{TMO}$ – trace of monitor observations of the tested monitor for monitor TM_ψ
EM - environment model
CE – controlled environment
OE – operational environment
ϕ – safety property
TC_ϕ - set of test cases for the system's safety property
$v^{O}_{MaxAssumed}$ - maximum assumed obstacles velocity
$v^{O}_{MaxObserved}$ - concrete monitor observation
$v^{O}_{MaxObserved}(\tau) = \max_{i>0, t\in[0,\tau]}(v_{O_i}(t))$
$v^{O_i}_{MaxTest}$ - max. test velocity of obstacle $O_i, i > 0$
$v^{O}_{MaxTest}$ - max. test velocity of any obstacle in the environment
$v^{O}_{MaxTest} = \max_{i>0}(v^{O_i}_{MaxTest})$
$v^{O_i}_{Max}$ - max. specific velocity of obstacle $O_i, i > 0$
v_{O_i} - current velocity of obstacle $O_i, i > 0$
$M \models \phi$ - – M is a model of the property specification ϕ or model M satisfies the property specification ϕ
✓ - qualitative assurance goal of the respective phase in the development process
✓ $\langle A \rangle$ - qualitative assurance goal fulfilled under the assumption that $\langle A \rangle$ is fulfilled
- part of the development process steps
- communication channels for data transfer

Figure 4.2.: Illustration of Solution Concept With Example Values.

phase is to *verify the safety requirement* ϕ. During system design, the technical system model is verified together with the environment model against the safety requirements specification under consideration of environment assumptions specification that are explicitly defined at design-time. Both the safety requirements specification as well as the environment assumptions are artifacts which result from the safety analysis carried out on the example system. The models SM and EM are executed in parallel during the design-time verification and exchange information with each other through a common interface. This interface corresponds to the parallel composition of two abstract models as defined in the literature (cf. [Hoa78], [SdV04]), which is then extended with a description of the environment assumptions. The parallel composition operator enhanced with the description of the environment assumptions is denoted by $||_{EA}$, where EA represents a set of environment assumptions defined at design-time. The description of the environment assumptions is given by a formal language, which is presented in Section 4.5.

The phase of system design is followed by the system test. During *system test*, the implemented system IS is run in parallel with the environment assumptions monitors in a controlled environment CE. Two quality assurance goals are set up for the system test phase: (1) ensure that the implemented system is a faithful implementation of the technical system model, e.g., by using back-to-back tests (cf. [AHDR18], [Lig09]), and (2) ensure that the implemented system behaves correctly with respect to its safety requirements, e.g., by using requirements-based test cases generated with the help of model checking (cf. Section 2.5.2). Additionally, the RMEA approach presented in this thesis introduces a third quality assurance goal: *test of the environment assumptions monitors* M_{EA}. The set of environment assumptions monitors run in parallel to the implemented system and observe the inputs received by the implemented system from the controlled environment. An alert mechanism is implemented in the monitors, which is triggered whenever a potentially dangerous situation is detected in the monitor observations. Since CE is a controlled environment, the test engineers can make specific changes to it, which would trigger the alert mechanism in the environment assumptions monitors. In this way, it can be checked whether changes in the controlled environment which contravene to the environment assumptions are detected by the environment assumptions monitors. Notice that the implemented system fulfills its safety requirement ϕ only if the monitor observations of the controlled environment CE fulfill the environment assumptions EA. This translates into the fact that the test cases TC_ϕ built from the safety requirement ϕ are passed only if each of the environment assumptions monitors $M_\psi \in M_{EA}$ evaluates to the truth value *true* on its respective trace of monitor observations $\pi_{MO}^{M_\psi}$.

The system test is followed by the requirements validation. Similar to the system test phase, there are specific quality assurance goals defined for the requirements validation phase. One goal is to validate the safety requirement specification ϕ, i.e., to ensure that the tested system TS satisfies the safety requirement ϕ. The RMEA approach introduces an additional quality assurance goal: *validation of the environment assumptions EA*. For this purpose, the tested system TS is set up in its operational environment OE, on the customer's site. The test engineers can use the same test cases TC_ϕ used during the system test phase. The validation of the environment assumptions is realized with the help of the set of environment assumptions monitors TM_{EA}. Notice that the safety requirement ϕ is valid only if every environment assumption in the set of environment assumptions EA is valid. This translates into the fact that the test cases TC_ϕ are passed only if every environment assumptions monitor $TM_\psi \in TM_{EA}$ evaluates to the truth value *true* on its respective trace of monitor observations π_{TMO}.

The approach proposed in this thesis is developed on the basis of three premises:

- the implemented system IS is a faithful implementation of the technical system model SM,
- the tested system TS preserves functionally the system behavior modeled in SM and implemented by IS, and
- the controlled environment CE is a faithful implementation of the environment model EM.

The first premise can be achieved by employing code generators which translate formal models into the corresponding source code of a programming language of choice, and at the same time, can guarantee the correctness of this transformation. For example, the qualified code generators of ASCET-DEVELOPER[1] or the ANSYS SCADE KCG [2] can translate finite state machines into C code. Methods of code generation from formal methods with guarantees for the correctness of the code transformation are out of scope of this thesis.

The second premise can be addressed during system test and the requirements validation phases through a separation of concerns between the system under test and any other artifacts which are necessary to carry out these phases of the development process. In the phase of system testing, the environment assumptions monitors M_{EA} are not allowed to change the functionality of the implemented system IS. The same considerations apply for the requirements validation phase with respect to the the tested environment assumptions monitors TM_{EA} and the tested system TS. This separation of concerns is important in order to preserve in the implemented system IS and respectively in the tested system TS the description of the system behavior which is conveyed through the technical system model SM. Before giving an example of change in the system's functionality, notice first that during the system test phase and respectively during the requirements validation phase, an alert mechanism implemented in the environment assumptions monitors is triggered whenever a violation of an environment assumption is detected in the controlled environment CE and respectively in the operational environment OE. An example of change in the system's functionality would be that the environment assumption monitor triggers a reconfiguration or adaptation of the system which is not defined in the technical system model SM. Methods of system reconfiguration and adaption that can be used to handle the alert message emitted by the environment assumptions monitors during system test and during requirements validation are out of the scope of this thesis and are not discussed further.

The third premise requires methods to translate and automatically integrate formal models of the environment model in a simulation environment. For specific application domains of autonomous safety-critical systems, e.g., automotive, controlled environments do not encompass only simulation environments, but also laboratory test stands and field test tracks. The realization of controlled environments such as test stands and test tracks in accordance with the environment model requires further methods, some of which are specific to the manufacturer of the autonomous safety-critical system. The complexity of the controlled environment depends on amount of resources which the system design team is willing to invest in its realization. Such methods are also excluded from the scope of this thesis.

4.1.3. Runtime Monitoring of Environment Assumptions by Example

In order to see how the proposed approach works concretely, let us consider as an example the mobile service robot introduced in Section 3.3 and go through every phase of the concept presented in Figure 4.2. As already mentioned in the previous section, this

[1] https://www.etas.com/en/products/ascet-developer.php

[2] https://www.ansys.com/products/embedded-software/ansys-scade-suite

concept is anchored in the system development process presented in Figure 4.1. Before starting with system design, requirements elicitation and analysis and safety analysis are performed in order to derive the functional requirements specification and the safety requirements specification for the mobile service robot. The result of the safety analysis is the safety requirements specification extended with environment assumptions. The safety requirement specification has been derived through safety analysis carried out on the mobile service robot in Section 3.4. For the sake of simplicity, the formal specification of the safety requirement defined in CTL in Section 3.4 is reiterated in Equation (4.1).

$$\phi : AG\ [\ (y_R == y_{O_i}) \wedge (x_{O_i} - x_R \geq -c_R) \wedge (x_{O_i} - x_R \leq c_R) \rightarrow (v_R == 0)\ ] \tag{4.1}$$

The safety requirement ϕ states that the mobile service robot must be stationary if any obstacle O_i is situated on its lane and inside the safety net spanned by the robot's reaction distance c_R. If the safety requirement is not fulfilled, then a collision between the robot and the obstacle O_i is imminent. How safety analysis can be used to derive environment assumption and how these assumptions extend the existing safety requirements specification is discussed in Section 4.4. For the purpose of this example, let us consider that the environment assumptions have been derived and explicitly defined. The rest of this section walks through every phase of the proposed concept and shows on the basis of the small example brought on in Figure 4.2 how the concept works.

System Design

The quality assurance goal of the system design phase is to verify the safety requirement ϕ. This is achieved with the help of design-time verification, specifically with model checking (cf. Section 2.5.2). Model checking is a formal verification method (cf. [Pel01]), which allows the automated analysis of dynamic systems that can be modeled by state-transition systems (cf. [CHV18]).

The input for the model checking procedure is a formal model of the system under analysis and a formal specification of the system requirement to be verified (cf. [BK08]). As already discussed in Section 3.5, the formal model given as input to the model checker consists in fact of two models, the technical system model and the environment model. The two models are combined through parallel composition with each other as depicted in Figure 4.2. The parallel composition of the two models is extended with a formal specification of the environment assumptions. In the example displayed in Figure 4.2, the technical system model SM uses the constant $v^{O}_{MaxAssumed} = 2.0$ to designate the assumed maximum obstacle velocity in the robot's environment. The environment model EM defines for each obstacle $O_i, i > 0$ the current velocity v_{O_i} and the maximum velocity $v^{O_i}_{Max}$, which is specific for each obstacle:

$$0.0 \leq v_{O_i} \leq v^{O_i}_{Max}, v^{O_i}_{Max} = 2.0, i \in \mathbb{N}_{\geq 1} \tag{4.2}$$

The set of environment assumptions EA contains only one environment assumption $\psi \in EA$, which is formulated with respect to the upper bound for the velocities of the obstacles which populate the robot's environment. The environment assumption ψ is defined by the predicate in Equation (4.3):

$$\psi : \forall i.\ v_{Max}^{O_i} \leq v_{MaxAssumed}^{O} \tag{4.3}$$

From the values used in the example during system design (cf. Figure 4.2), it is fairly easy to notice that the environment assumption in Equation (4.3) is valid, and as such, the safety requirement specification ϕ is satisfied.

The formal specification of the explicitly defined environment assumptions is used at design-time to build the environment assumptions monitors. To each environment assumption there is a corresponding environment assumptions monitor. Thus, the set of environment assumptions monitors M_{EA} contains the environment assumptions monitor $M_\psi \in M_{EA}$, defined in Equation (4.4), which corresponds to the environment assumption ψ introduced in Equation (4.3):

$$M_\psi : v_{MaxObserved}^{O} \leq v_{MaxAssumed}^{O} \tag{4.4}$$

Notice that the environment assumptions monitor M_ψ compares two variables $v_{MaxAssumed}^{O}$ and $v_{MaxObserved}^{O}$. During system test and requirements validation, the environment assumptions monitor is evaluated throughout the entire execution of the system under analysis. Therefore, the two variables $v_{MaxAssumed}^{O}$ and $v_{MaxObserved}^{O}$ can be regarded as functions over the system's execution steps. The first one, $v_{MaxAssumed}^{O}$, is defined as a constant function in the technical system model SM:

$$\begin{aligned} v_{MaxAssumed}^{O} &: \mathbb{N} \to \mathbb{R}_{\geq 0} \\ v_{MaxAssumed}^{O}(\tau) &= 2.0, \forall\ \tau \in \mathbb{N} \end{aligned} \tag{4.5}$$

while the second one, $v_{MaxObserved}^{O}$, is a function introduced by the environment assumption monitor M_ψ. Notice that $v_{MaxObserved}^{O}$ describes the maximum observed obstacle velocity for any visible obstacle in the robot's environment, which varies throughout the system execution:

$$\begin{aligned} v_{MaxObserved}^{O} &: \mathbb{N} \to \mathbb{R}_{\geq 0} \\ v_{MaxObserved}^{O}(\tau) &= \max_{i>0, t\in[0,\tau]} (v_{O_i}(t)), \forall \tau \in \mathbb{N} \end{aligned} \tag{4.6}$$

System Test

In the system test phase, the quality assurance goal introduced by the RMEA approach is the test of the environment assumptions monitors M_{EA}. The implemented system IS maintains the assumed maximum velocity at $v_{MaxAssumed}^{O} = 2.0$ as defined in system design, while in the controlled environment CE, the upper limit of the current obstacle velocity v_{O_i} is given by a maximum test velocity $v_{MaxTest}^{O_i}$, that is defined by the test engineers:

$$0.0 \leq v_{O_i} \leq v_{MaxTest}^{O_i}, i \in \mathbb{N}_{\geq 1} \tag{4.7}$$

The environment assumptions monitor is evaluated at every execution step $\tau \in \mathbb{N}$ during the system test of the implemented system, as shown in Equation (4.8):

$$M_\psi[\tau] \overset{\text{def}}{=} v^O_{MaxObserved}(\tau) \leq v^O_{MaxAssumed}(\tau) \tag{4.8}$$

Since it executes parallel to the implemented system in the controlled environment, the environment assumptions monitor M_ψ is constantly evaluated against a finite trace of monitor observations of the form shown in Equation (4.9):

$$\pi^{M_\psi}_{MO} = (v^O_{MaxObserved}(0), v^O_{MaxObserved}(1), \ldots, v^O_{MaxObserved}(\tau - 1), v^O_{MaxObserved}(\tau)) \tag{4.9}$$

The corresponding monitor trace π_{M_ψ} of the environment assumptions monitor M_ψ is shown in Equation (4.10):

$$\begin{aligned} \pi_{M_\psi} =& ((v^O_{MaxObserved}(0), v^O_{MaxAssumed}(0)), (v^O_{MaxObserved}(1), v^O_{MaxAssumed}(1)) \\ & \ldots, (v^O_{MaxObserved}(\tau), v^O_{MaxAssumed}(\tau))) \end{aligned} \tag{4.10}$$

During system test, the upper bound of the current obstacle velocity v_{O_i} is $v^{O_i}_{MaxTest}$ for an obstacle $O_i, i \in \mathbb{N}_{\geq 1}$ (cf. Equation (4.7)). The upper bound for $v^O_{MaxObserved}(\tau), \tau > 0$ is denoted $v^O_{MaxTest}$:

$$0.0 \leq v^O_{MaxObserved}(\tau) \leq v^O_{MaxTest} \tag{4.11}$$

and can be derived from the Equations (4.6) and (4.7) as:

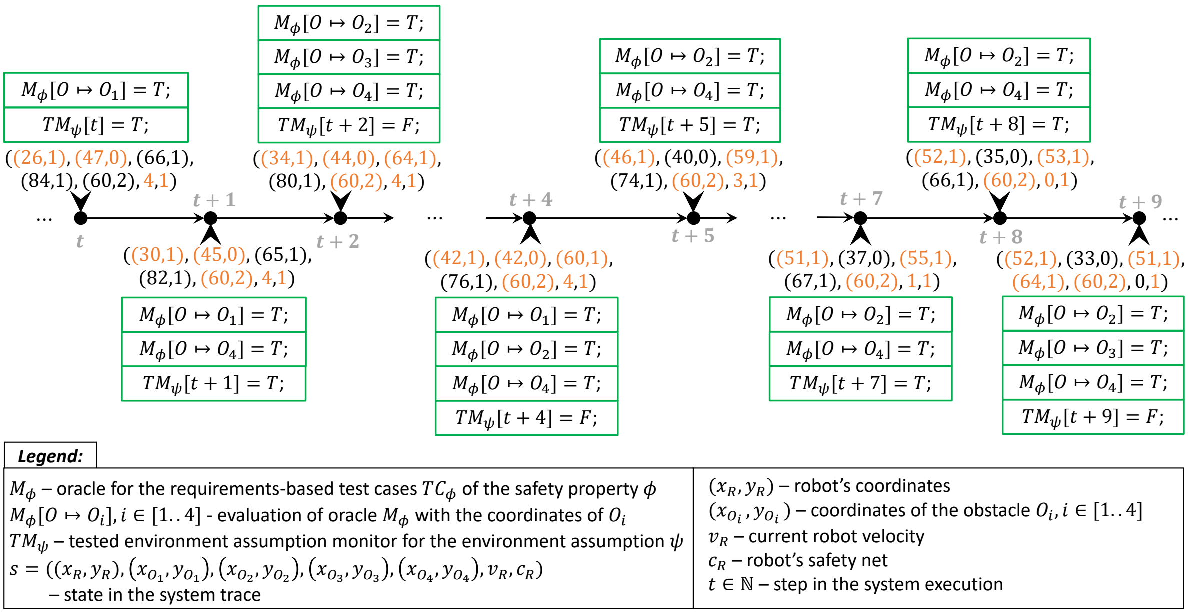
$$v^O_{MaxTest} \overset{\text{def}}{=} \max_{i \geq 1}(v^{O_i}_{MaxTest}) \tag{4.12}$$

The system test phase can be used by the test engineers in a two-fold manner. On one side, the test engineers can develop requirements-based test cases in order to check the implemented system with respect to its safety requirement. On the other side, the test engineers can design test cases to test the environment assumptions monitor. Notice that the environment assumption ψ is defined explicitly at design-time. The test engineers are aware of it and can choose $v^{O_i}_{MaxTest}$ such that a violation of the environment assumption is forced and test if environment assumptions monitor M_ψ is triggered by this violation.

Consider the example illustrated in Figure 4.2, specifically the values used in the system test phase. By choosing $v^{O_i}_{MaxTest} \in \{1.0, 1.5, 2.0\}$ the test engineers create test cases to show that the implemented system IS fulfills the safety property ϕ. Thus, if the maximum test velocity is maintained at $v^{O_i}_{MaxTest} \leq 2.0$ for all obstacles O_i, then the monitor observation trace π_{MO} satisfies the environment assumptions monitor $M_\psi \in M_{EA}$, i.e., $\pi^{M_\psi}_{MO} \models M_\psi$. By the results of the design-time verification, the safety requirement specification ϕ is also satisfied. On the other hand, test engineers can design test cases in order to target and test specifically the environment assumptions monitors defined in system design. By choosing $v^{O_i}_{MaxTest} \in \{2.5, 3.0, 3.5\}$ it is possible to build a monitor observations trace that does not satisfy the environment assumptions monitor $M_\psi \in M_{EA}$. Once it is triggered, the environment assumptions monitor M_ψ sends an alert message to the implemented system IS.

Requirements Validation

In the requirements validation phase, the quality assurance goal introduced by the RMEA approach is the validation of the environment assumptions EA. This is achieved with the help of the set of tested environment assumptions monitors TM_{EA}. Requirements validation shows that the tested system TS fulfills the safety requirement ϕ under the premise that the set of environment assumptions EA are valid. However, this depends on the maximum velocities of the obstacles in the robot's operational environment. These velocities are unknown during the robot's operation time. The consequence is that, if the environment assumptions are not valid, then no statement can be made as to whether the safety requirement of the mobile service robot is satisfied or not. In order to demonstrate this, consider the system trace in Figure 4.3, which also depicts the traces of the environment assumption monitor $TM_\psi \in TM_{EA}$ and the trace of the test oracle M_ϕ. Both the environment assumption monitor TM_ψ and the test oracle M_ϕ are evaluated only with respect to the obstacles visible inside the robot's sensor horizon. In order to simplify the analysis and maintain consistency with the analysis carried out in Chapter 3, the robot's sensor horizon is set up at $h_R = 30$ distance units, while the robot's safety net is considered to be constant and as wide as one distance unit, i.e., $c_R = 1$.



$M_\phi[O \mapsto O_1] = T;$
$TM_\psi[t] = T;$
((26,1), (47,0), (66,1), (84,1), (60,2), 4,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_3] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+2] = F;$
((34,1), (44,0), (64,1), (80,1), (60,2), 4,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+5] = T;$
((46,1), (40,0), (59,1), (74,1), (60,2), 3,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+8] = T;$
((52,1), (35,0), (53,1), (66,1), (60,2), 0,1)
…
t
$t+1$
$t+2$
…
$t+4$
$t+5$
…
$t+7$
$t+8$
$t+9$
…
((30,1), (45,0), (65,1), (82,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+1] = T;$
((42,1), (42,0), (60,1), (76,1), (60,2), 4,1)
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+4] = F;$
((51,1), (37,0), (55,1), (67,1), (60,2), 1,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+7] = T;$
((52,1), (33,0), (51,1), (64,1), (60,2), 0,1)
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_3] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+9] = F;$
Legend:
M_ϕ – oracle for the requirements-based test cases TC_ϕ of the safety property ϕ
$M_\phi[O \mapsto O_i], i \in [1..4]$ - evaluation of oracle M_ϕ with the coordinates of O_i
TM_ψ – tested environment assumption monitor for the environment assumption ψ
$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), (x_{O_2}, y_{O_2}), (x_{O_3}, y_{O_3}), (x_{O_4}, y_{O_4}), v_R, c_R)$
– state in the system trace
(x_R, y_R) – robot's coordinates
(x_{O_i}, y_{O_i}) – coordinates of the obstacle $O_i, i \in [1..4]$
v_R – current robot velocity
c_R – robot's safety net
$t \in \mathbb{N}$ – step in the system execution

Figure 4.3.: Mobile Service Robot: Application of RMEA to a System Trace with **valid Environment Assumption** ψ and **valid Safety Requirement** ϕ.

Notice that the environment assumption monitor $TM_\psi \in TM_{EA}$ evaluates to the truth value *true* on the trace depicted in Figure 4.3, i.e., the maximum observed obstacle velocity $v^O_{MaxObserved}$ is never larger than the maximum assumed obstacle velocity $v^O_{MaxAssumed}$. Furthermore, the test oracle M_ϕ also evaluates to *true*, which means that the safety requirement ϕ is valid for the trace shown in Figure 4.3. Remember that the safety requirement demands that the robot is stationary if there is an obstacle situated on the ego lane and inside the robot's safety net. In order to see this, notice the system states at the steps $t + 8$ and $t + 9$, with $t \in \mathbb{N}$. At step $t + 8$ the robot is stationary at coordinates $(52, 1)$ and obstacle O_2 is situated on the ego lane inside the robot's safety net at position $(53, 1)$. At step $t + 9$, obstacle O_2 passes by the robot arriving at position $(51, 1)$. Notice that at execution step $t + 9$ obstacle O_2 is still situated inside the safety net of the robot, which is why the robot remains stationary at position $(52, 1)$. According to the system requirements defined in Section 3.3, the robot eventually picks up his drive towards his destination as soon as obstacle O_2 has exited its safety net.

Environment assumptions defined at design-time may be invalid at the robot's operation time or become invalid as the robot progresses with its drive towards its destination. An example of an invalid environment assumption is shown in Figure 4.4. The question is what can be said about the validity of the safety requirement in case the environment assumptions are invalid. For this purpose consider the system and monitor trace depicted in Figure 4.4. At step $t + 1$ the environment assumptions monitor TM_{ψ} is evaluated to the truth value *false* because the current velocity of the obstacle O_1, situated on the left lane, is $v_{O_1} = 3.0$, and thus exceeds the maximum assumed velocity $v^{O}_{MaxAssumed} = 2.0$. Once the environment assumptions monitor is triggered, an alert message is sent to the system, which can react in a variety of ways. One possible reaction of the mobile service robot is to start braking. At execution step $t + 2$, obstacle O_2, situated on the ego lane, also accelerates to velocity $v_{O_2} = 3.0$ and maintains this velocity until step $t + 4$, when it increases again its velocity up to $v_{O_2} = 4.0$. Despite this, the robot is able to brake to a standstill at step $t + 4$ and thus avoid an active collision and fulfill its safety requirement. This is because the robot has started to brake as soon as the environment assumptions monitor TM_{ψ} has been triggered by the behavior of obstacle O_1.

Consider now the system and monitor trace depicted in Figure 4.5. This trace features at time point t the same system configuration as the trace displayed in Figure 4.4. Similar to the trace in Figure 4.4, the trace in Figure 4.5 depicts obstacle O_1 on the left lane as it increases its speed at step $t + 1$ to $v_{O_1} = 3.0$. In turn, obstacle O_2, situated on the ego lane, increases its velocity at step $t + 2$ to $v_{O_2} = 3.0$ and maintains it until execution step $t + 3$. Despite obstacle O_2 decreasing its velocity to $v_{O_2} = 2.0$ at step $t + 4$, the robot causes an active collision with obstacle O_2. The robot violates its safety requirement ϕ, i.e., for obstacle O_2 the test oracle M_{ϕ} is evaluated to the truth value *false*. The violation of the safety requirement ϕ happens because the robot does not start to brake as soon as the environment assumption becomes invalid at step $t + 1$. Instead, the robot begins to brake one step later at execution step $t + 2$, as soon as it perceives obstacle O_2 inside its collision distance. The brake maneuver carried out by the robot at step $t + 2$ turns out to be too late for the avoidance of an active collision.

It is worth noticing that the approach proposed in this thesis is applicable at any hierarchy level in the system under analysis, independent of the granularity of the test object. Remember that the test object can be either a software system component, a subsystem or even the whole system, while the environment of the test object is

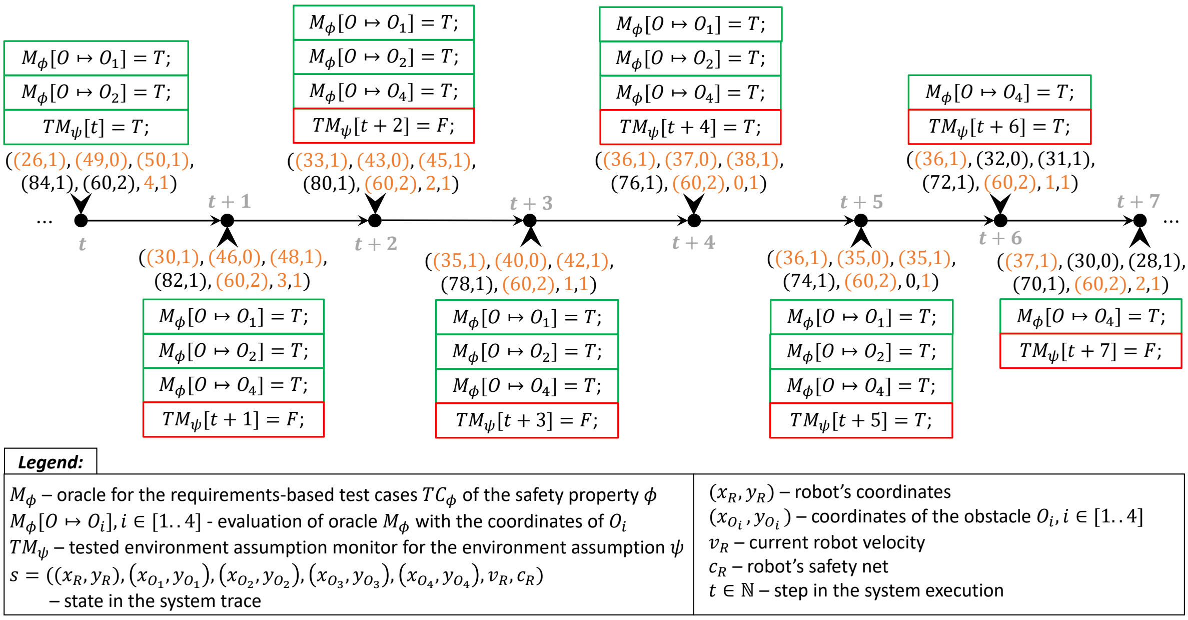

$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$TM_\psi[t] = T;$
((26,1), (49,0), (50,1), (84,1), (60,2), 4,1)
t
((30,1), (46,0), (48,1), (82,1), (60,2), 3,1)
$t+1$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+1] = F;$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+2] = F;$
((33,1), (43,0), (45,1), (80,1), (60,2), 2,1)
$t+2$
((35,1), (40,0), (42,1), (78,1), (60,2), 1,1)
$t+3$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+3] = F;$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+4] = T;$
((36,1), (37,0), (38,1), (76,1), (60,2), 0,1)
$t+4$
((36,1), (35,0), (35,1), (74,1), (60,2), 0,1)
$t+5$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+5] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+6] = T;$
((36,1), (32,0), (31,1), (72,1), (60,2), 1,1)
$t+6$
((37,1), (30,0), (28,1), (70,1), (60,2), 2,1)
$t+7$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+7] = F;$
...
Legend:
M_ϕ – oracle for the requirements-based test cases TC_ϕ of the safety property ϕ
$M_\phi[O \mapsto O_i], i \in [1..4]$ - evaluation of oracle M_ϕ with the coordinates of O_i
TM_ψ – tested environment assumption monitor for the environment assumption ψ
$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), (x_{O_2}, y_{O_2}), (x_{O_3}, y_{O_3}), (x_{O_4}, y_{O_4}), v_R, c_R)$ – state in the system trace
(x_R, y_R) – robot's coordinates
(x_{O_i}, y_{O_i}) – coordinates of the obstacle $O_i, i \in [1..4]$
v_R – current robot velocity
c_R – robot's safety net
$t \in \mathbb{N}$ – step in the system execution

Figure 4.4.: Mobile Service Robot: Application of RMEA to a System Trace with **invalid Environment Assumption** ψ and **valid Safety Requirement** ϕ.

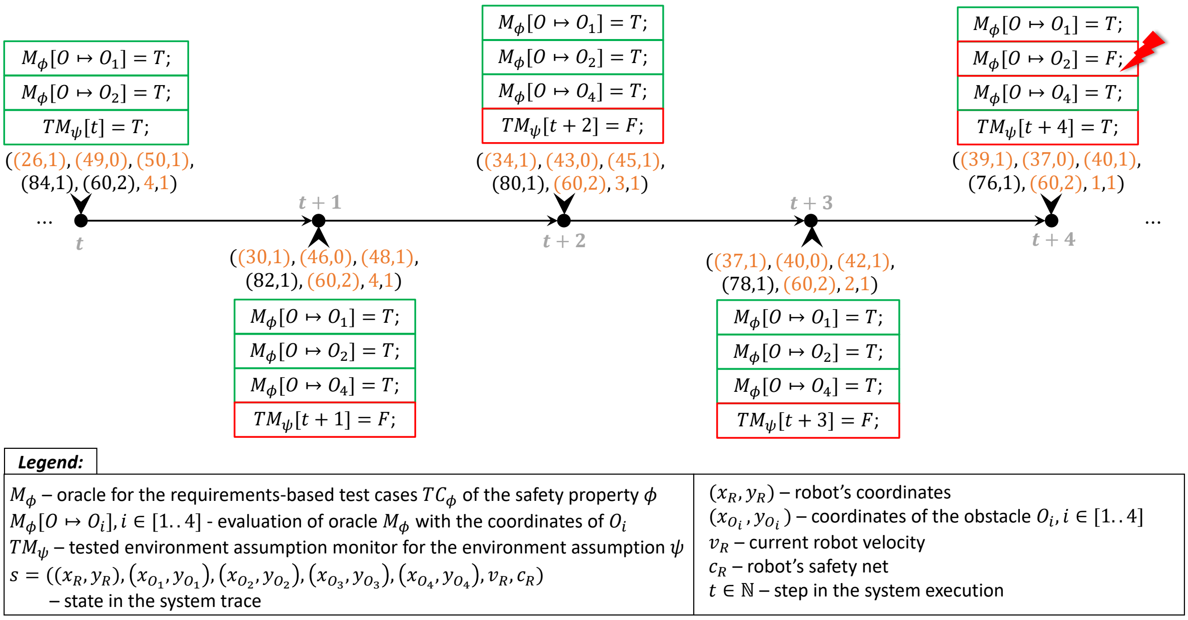

$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$TM_\psi[t] = T;$
((26,1), (49,0), (50,1), (84,1), (60,2), 4,1)
t
((30,1), (46,0), (48,1), (82,1), (60,2), 4,1)
$t+1$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+1] = F;$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+2] = F;$
((34,1), (43,0), (45,1), (80,1), (60,2), 3,1)
$t+2$
((37,1), (40,0), (42,1), (78,1), (60,2), 2,1)
$t+3$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = T;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+3] = F;$
$M_\phi[O \mapsto O_1] = T;$
$M_\phi[O \mapsto O_2] = F;$
$M_\phi[O \mapsto O_4] = T;$
$TM_\psi[t+4] = T;$
((39,1), (37,0), (40,1), (76,1), (60,2), 1,1)
$t+4$
...
Legend:
M_ϕ – oracle for the requirements-based test cases TC_ϕ of the safety property ϕ
$M_\phi[O \mapsto O_i], i \in [1..4]$ - evaluation of oracle M_ϕ with the coordinates of O_i
TM_ψ – tested environment assumption monitor for the environment assumption ψ
$s = ((x_R, y_R), (x_{O_1}, y_{O_1}), (x_{O_2}, y_{O_2}), (x_{O_3}, y_{O_3}), (x_{O_4}, y_{O_4}), v_R, c_R)$ – state in the system trace
(x_R, y_R) – robot's coordinates
(x_{O_i}, y_{O_i}) – coordinates of the obstacle $O_i, i \in [1..4]$
v_R – current robot velocity
c_R – robot's safety net
$t \in \mathbb{N}$ – step in the system execution

Figure 4.5.: Mobile Service Robot: Application of RMEA to a System Trace with **invalid Environment Assumption** ψ and **invalid Safety Requirement** ϕ.

essentially a combination of a technical and physical environment, depending on the level of abstraction at which the test object is situated.

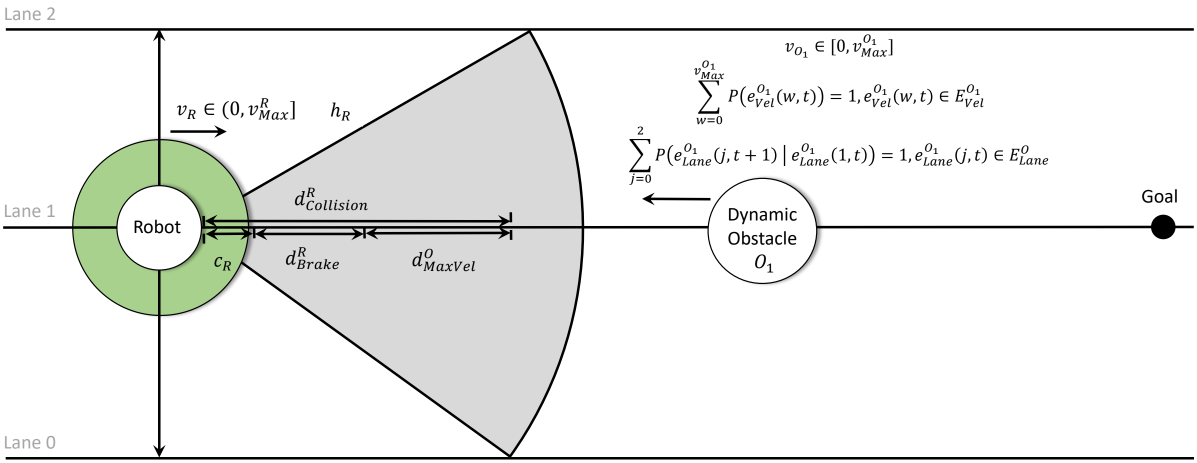
4.2. Revisiting the Motivational Example

Chapter 3 presented the first version of the motivational example as a basis for the problem analysis of this thesis. The motivational example is centered around the a mobile service robot which moves autonomously in an uncertain environment. Several constraints were formulated with respect to the robot's environment, in order to simplify the problem analysis. The environment is represented as a subset of the Cartesian plane, in which the robot and the static and dynamic obstacles in its environment are approximated to discrete points. Each obstacle O_i is characterized by its current velocity v_{O_i} and its current position (x_{O_i}, y_{O_i}). The current velocity of each obstacle O_i has an upper bound represented by its specific maximum velocity $v_{Max}^{O_i}$. Thus, for any obstacle O_i, the following holds: $v_{O_i} \in [0.0, v_{Max}^{O_i}]$. Furthermore, the first version of the motivational example presented in Chapter 3 put very specific constraints on the behavior of the obstacles situated in the environment of the mobile service robot. Thus, dynamic obstacles were allowed to move forwards and overcome slower obstacles by driving through them. However, dynamic obstacles were forbidden to change lanes, move backwards, or drive in the same direction as the robot if they were situated in one of its blind spots. Moreover, dynamic obstacles which have become stationary were not allowed to resume their movement.

This section revisits the motivational example and relaxes some of these constraints, while other constraints are kept in place. Firstly, dynamic obstacles are allowed to change lanes. Notice that a dynamic obstacle $O_i, i \in \mathbb{N}_{\geq 1}$ can change to another lane if it wants to overcome a slower or a stationary obstacle $O_j, j \neq i, i \in N_{\geq 1}$. Furthermore, stationary obstacles are allowed to start moving, while dynamic obstacles which have become stationary are allowed to resume their movement. The constraints that remain unchanged are that dynamic obstacles are forbidden to move backwards and to move in the same direction as the robot if they are situated in one of its blind spots.

A physical overview of the revisited motivational example is given in Figure 4.6. In order to simplify the analysis, the environment of the mobile service robot is populated by one obstacle, denoted O_1. Obstacle O_1 is able to move forwards, increase and reduce its velocity, i.e., accelerate and brake, come to a stop, and then resume its movement again.

With respect to lane changing, there are several rules are introduced that obstacle O_1 has to abide by, which are illustrated in Figure 4.7. The cases 1 to 4 illustrate lane changes permitted for obstacle O_1, while cases 5 and 6 depict forbidden lane changes. Thus, if obstacle O_1 is situated on one of the outer lanes - lane 0 or lane 2 - at step t, then it is allowed to change to the middle lane, i.e., lane 1, at step $t + 1$. The opposite is also permitted, i.e., if obstacle O_1 is situated on the middle lane at step t, then it can change to one of the outer lanes at step $t + 1$. However, obstacle O_1 is not allowed to


Lane 2
$v_{O_1} \in [0, v_{Max}^{O_1}]$
$\sum_{w=0}^{v_{Max}^{O_1}} P\big(e_{Vel}^{O_1}(w,t)\big) = 1, e_{Vel}^{O_1}(w,t) \in E_{Vel}^{O_1}$
$\sum_{j=0}^{2} P\big(e_{Lane}^{O_1}(j,t+1) \mid e_{Lane}^{O_1}(1,t)\big) = 1, e_{Lane}^{O_1}(j,t) \in E_{Lane}^{O}$
$v_R \in (0, v_{Max}^R]$
h_R
Lane 1
Robot
$d_{Collision}^R$
c_R
d_{Brake}^R
d_{MaxVel}^O
Dynamic Obstacle O_1
Goal
Lane 0

Legend:

$d_{Collision}^R$ - collision distance of the robot
d_{Brake}^R - emergency braking distance of the robot, in relation of the robot's current velocity at time
c_R – robot's safety net (dependent on the robot's processing time)
h_R – visual horizon of the robot's sensors
v_R - current robot velocity
v_{Max}^R - maximum robot velocity

d_{MaxVel}^O - maximum distance covered by the visible obstacle on the robot's lane if it moves with the maximum velocity assumed for any dynamic obstacle during the robot's processing and braking time
v_{O_1} - current velocity of obstacle O_1
$v_{Max}^{O_1}$ - maximum velocity of the obstacle O_1
$E_{Vel}^{O_1}$ - set of events with respect to changes in the obstacles velocity
$e_{Vel}^{O_1}(w,t)$ - event representing that obstacle O_1has the velocity w at step t
E_{Lane}^{O}- set of events with respect to obstacle lane changes
$e_{Lane}^{O_1}(j,t)$ - event representing that obstacle O_1 is situated on lane j at step t, with $j \in [0, n_{Lanes}-1]$ and $n_{Lanes}=3$ the number of lanes

Figure 4.6.: Physical Overview of the Revisited Motivational Example.

jump over several lanes between two consecutive steps, i.e., if obstacle O_1 is situated on lane 0 at step t, then it cannot change directly to lane 2 at step $t+1$ and vice versa.

Remember that in case of collision danger, the original version of the motivational example presented in Section 3.1 allows the robot to change to a lane situated farther away if it detects it to be a safe target lane. It is worth noticing that the lane changing rules defined in Figure 4.7 for the obstacles in the robot's environment are also imposed for the robot itself. With the introduction of the lane changing rules, in case of collision danger the robot checks whether the left lane or the right lane are safe target lanes. In case one of these lanes is safe, then the robot executes a lane change. Otherwise, the robot keeps its lane.

Notice that the behavior of obstacle O_1 is inherently uncertain. Consider for example first case and second case in Figure 4.7 in which the dynamic obstacle O_1 drives on one of the outer lanes, lane 0 or respectively lane 2, and is allowed to change to lane 1. Although obstacle O_1 has the possibility to move to lane 1, it may choose to continue its movement on its current lane. A similar way of thinking applies to the third and fourth case in Figure 4.7, where obstacle O_1 has three choices, namely staying on its current lane, changing to the left lane, or changing to the right lane. With respect to the fifth and sixth case depicted in Figure 4.7, one would think that given the rules imposed on lane changing in the robot's environment, the only choice for obstacle O_1 is to maintain its current lane. However, it is possible that obstacle O_1 chooses to ignore these rules

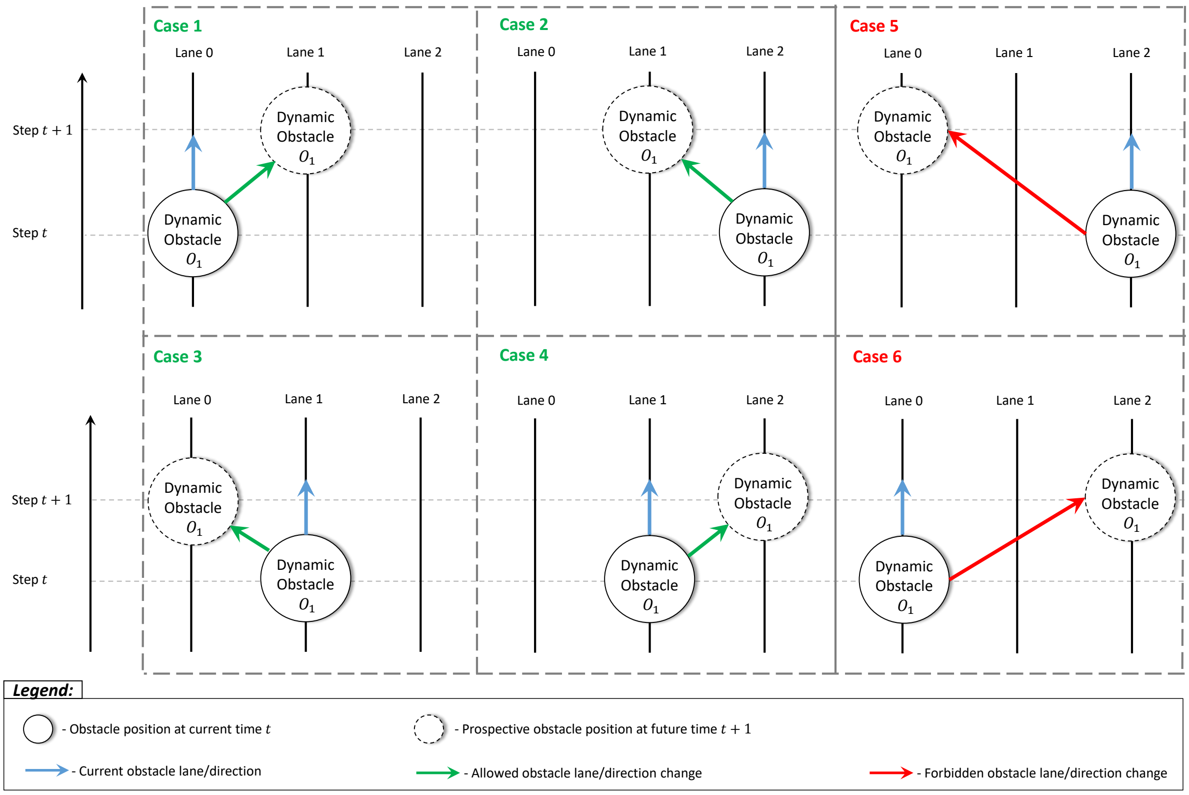

Case 1
Case 2
Case 5
Case 3
Case 4
Case 6
Lane 0
Lane 1
Lane 2
Step $t+1$
Step t
Dynamic Obstacle O_1
Legend:
- Obstacle position at current time t
- Prospective obstacle position at future time $t+1$
- Current obstacle lane/direction
- Allowed obstacle lane/direction change
- Forbidden obstacle lane/direction change

Figure 4.7.: Overview of the Rules for Lane Changing in the Revisited Motivational Example.

and jump over two lanes between two consecutive steps, thus executing an illegal lane change.

Environment uncertainty can be modeled using the probabilities theory. Several notations are introduced to represent the uncertainty in the lane changing maneuvers of the visible obstacles in the robot's environment. These notations are considered to be effective for the remainder of this work.

Notation 4.2.1. This notation introduces the sample space, the event space and the probability function related to the lane changing maneuvers of the obstacles in environment of the mobile service robot. Let $i \in \mathbb{N}_{\geq 1}$ be a natural number which indexes through the obstacles in the robot's environment.

1. The sample space for the lane changing maneuvers is the set of all possible outcomes of a lane change of the obstacles and is denoted by Ω_{Lane}:

$$\Omega_{Lane} \overset{\text{def}}{=} \{0, 1, \dots, n_{Lanes} - 1\} \tag{4.13}$$

where $n \in \mathbb{N},\ n \geq 1$ is the number of lanes.

2. The event space for the lane changing maneuvers is a set of events in the sample space Ω_{Lane} and is denoted by E^{O}_{Lane}:

$$E^{O}_{Lane} \overset{\text{def}}{=} \{e^{O_i}_{Lane}(j) \mid j \in \Omega_{Lane}, i \in \mathbb{N}_{\geq 1}\} \tag{4.14}$$

3. A single event in the event space E^{O}_{Lane} is represented by a set of outcomes in the sample space Ω_{Lane} and is denoted by $e^{O_i}_{Lane}(j) \subset \Omega_{Lane}$.
4. The event $e^{O_i}_{Lane}(j)$ denotes that the visible obstacle O_i is situated on lane $j \in \Omega_{Lane}$:

$$e^{O_i}_{Lane}(j) \overset{\text{def}}{=} (y_{O_i} = j) \tag{4.15}$$

5. Each event in the set E^{O}_{Lane} is observed over the sequence of steps in the system execution. Therefore, it is possible to talk about an event taking place at step t, which is denoted by $e^{O_i}_{Lane}(j, t)$:

$$e^{O_i}_{Lane}(j, t) \overset{\text{def}}{=} (y_{O_i}(t) = j) \tag{4.16}$$

■

It is worth noticing that the rules for lane changing depicted in Figure 4.7 introduce conditionality between the events in set E^{O}_{Lane}, which also poses the need for conditional probabilities. For the motivational example revisited in Figure 4.6 there are three cases to be considered in which conditional probabilities for lane changing can be computed:

1. Obstacle O_1 moves on lane 0 at step t. The legal moves which obstacle O_1 can execute at step $t+1$ are to keep moving on lane 0 or change to lane 1. The change to lane 2 is illegal according to the rules introduced in Figure 4.7:

$$\begin{aligned} P(e^{O_1}_{Lane}(0, t+1) \mid e^{O_1}_{Lane}(0, t)) +\\ P(e^{O_1}_{Lane}(1, t+1) \mid e^{O_1}_{Lane}(0, t)) +\\ P(e^{O_1}_{Lane}(2, t+1) \mid e^{O_1}_{Lane}(0, t)) = 1 \end{aligned} \tag{4.17}$$

2. Obstacle O_1 moves on lane 1 at step t. The legal moves which obstacle O_1 can carry out at step $t+1$ are to remain on lane 1 or change to lane 0 or to lane 2.

$$\begin{aligned} P(e^{O_1}_{Lane}(1, t+1) \mid e^{O_1}_{Lane}(1, t)) +\\ P(e^{O_1}_{Lane}(0, t+1) \mid e^{O_1}_{Lane}(1, t)) +\\ P(e^{O_1}_{Lane}(2, t+1) \mid e^{O_1}_{Lane}(1, t)) = 1 \end{aligned} \tag{4.18}$$

3. Obstacle O_1 moves on lane 2 at step t. The legal moves which obstacle O_1 can execute at step $t+1$ are to maintain the movement on lane 2 or change to lane 1. The change to lane 0 is illegal according to the rules introduced in Figure 4.7:

$$\begin{aligned} P(e^{O_1}_{Lane}(2, t+1) \mid e^{O_1}_{Lane}(2, t)) +\\ P(e^{O_1}_{Lane}(1, t+1) \mid e^{O_1}_{Lane}(2, t)) +\\ P(e^{O_1}_{Lane}(0, t+1) \mid e^{O_1}_{Lane}(2, t)) = 1 \end{aligned} \tag{4.19}$$

In general, if a visible obstacle $O_i, i \in \mathbb{N}_{\geq 1}$ is situated on lane $k \in \{0, \ldots, n-1\}, n \in \mathbb{N}_{\geq 1}$ then the conditional probabilities $P(e^{O_i}_{Lane}(j, t+1) \mid e^{O_i}_{Lane}(k, t))$ with $j \in \{0, \ldots, n-1\}$ are computed at each execution step t. Naturally, the sum of these probabilities always amounts to 1:

$$\sum_{j=0}^{n-1} P(e^{O_i}_{Lane}(j, t+1) \mid e^{O_i}_{Lane}(k, t)) = 1 \tag{4.20}$$

The mobile service robot works on the premise that obstacle O_1 abides by the lane changing rules presented in Figure 4.7. In order to represent this and also to account for the possibility of obstacle O_1 executing an illegal lane change, the probabilities of illegal lane changes can be set to be lower than the probabilities of legal lane changes.

Besides the lane changing behavior of obstacle O_1, another source of uncertainty is related to its velocity, since obstacle O_1 can accelerate, brake, come to a halt and then start its movement again. Several notations are introduced to represent the uncertainty in the velocity changes of the visible obstacles in the robot's environment. These notations are considered to be effective for the remainder of this work.

Notation 4.2.2. This notation introduces the sample space, the event space and the probability function related to the velocity changing maneuvers, i.e., acceleration, braking, or stopping, of the obstacles in environment of the mobile service robot. Let $i \in \mathbb{N}_{\geq 1}$ be a natural number to index through the visible obstacles in the robot's environment.

1. The sample space for the velocity changing maneuvers of obstacle O_i is the set of all possible outcomes of a change in velocity for obstacle O_i and is denoted by $\Omega_{Vel}^{O_i}$:

$$\Omega_{Vel}^{O_i} \overset{\text{def}}{=} [0.0, v_{Max}^{O_i}] \tag{4.21}$$

2. The event space for the velocity changing maneuvers is a set of events in the sample space $\Omega_{Vel}^{O_i}$ and is denoted by $E_{Vel}^{O_i}$:

$$E_{Vel}^{O_i} \overset{\text{def}}{=} \{e_{Vel}^{O_i}(w) \mid w \in \Omega_{Vel}^{O_i}, i \in \mathbb{N}_{\geq 1}\} \tag{4.22}$$

3. A single event in the event space $E_{Vel}^{O_i}$ is represented by a set of outcomes in the sample space $\Omega_{Vel}^{O_i}$ and is denoted by $e_{Vel}^{O_i}(w) \subset \Omega_{Vel}^{O_i}$.
4. The event $e_{Vel}^{O_i}(w)$ denotes that the visible obstacle O_i has its current velocity equal to $w \in \Omega_{Vel}^{O_i}$

$$e_{Vel}^{O_i}(w) \overset{\text{def}}{=} (v_{O_i} = w) \tag{4.23}$$

5. Each event in the set $E_{Vel}^{O_i}$ is observed over the system execution. Therefore it is possible to talk about an event taking place at execution step t, which is denoted by $e_{Vel}^{O_i}(w, t)$:

$$e_{Vel}^{O_i}(w, t) \overset{\text{def}}{=} (v_{O_i}(t) = w) \tag{4.24}$$

■

In general, for the visible obstacle $O_i, i \in \mathbb{N}_{\geq 1}$ the probabilities $P(e_{Vel}^{O_i}(w, t+1))$ with $w \in [0, v_{Max}^{O_i}]$ are computed at each step t. Naturally, the sum of these probabilities always amounts to 1:

$$\sum_{w=0}^{v_{Max}^{O_i}} P(e_{Vel}^{O_i}(w, t+1)) = 1 \tag{4.25}$$

With the notations introduced in in this section, the velocity and lane changing maneuvers carried out over time by obstacle O_1 are described by probability distributions, which express the probability of being on a given lane or moving with a given velocity for obstacle O_1. Theoretically, the execution of the mobile service robot is infinite or can strech over a very long period of time, which makes computing probability distribution over the entire system execution unfeasible. Therefore, system designers can establish time windows of a specific length during which the probability distributions for velocity and lane changing maneuvers can be computed.

Notice that the current velocity of the obstacle as well as the lane on which the obstacle maneuvers are modeled as discrete random variables. Before discussing the modeling of the obstacle lane and obstacle velocity as discrete random variables, it is worth looking at how their respective sample spaces are defined. While for lane changes, the sample space and the event space remain unchanged for every obstacle in the robot's environment, for events describing obstacle velocity changes the sample space and respectively the event space are specific to each obstacle in the robot's environment. This is because all obstacles in the robot's environment have the same lanes at their disposal, i.e., the lanes identification numbers do not change, while for each obstacle there is a specific maximum velocity $v_{Max}^{O_i}$, that may be different for every obstacle. The representation of the obstacle's lane as a discrete random variable does not raise any issues as there is a finite number of lanes and the lanes can be counted and respectively uniquely identified by their assigned identification number. However, the modeling of the obstacle's velocity as a discrete random variable merits a more extensive discussion. The current velocity v_{O_i} of an obstacle O_i is situated in the interval $[0.0, v_{Max}^{O_i}]$. As a discrete variable, the current velocity v_{O_i} of obstacle O_i changes throughout the system execution by taking discrete values from the interval $[0.0, v_{Max}^{O_i}]$. A constant offset Δ can be used in order to discretize the velocity interval. The offset Δ can be chosen in such a way that it reflects also the desired precision in the decimal point for the current obstacle velocity. For the example of the mobile service robot an offset of $\Delta = 1$ has been chosen. Due to the discretization, the obstacle velocity takes a finite number of values. This will prove to be useful later in the system design for the purpose of design-time verification via model checking, as the system and the environment can be represented by finite state models, thus simplifying considerably their analysis during design-time. An approach closer to reality is representing the obstacle velocity as a continuous random variable. In this case the obstacle velocity can take an infinite number of values from the interval $[0.0, v_{Max}^{O_i}]$. This translates to the system and the environment models having an infinite state space, which may lead to the state space explosion problem during the design-time verification via model checking.

The behavior of the mobile service robot remains simple. The robot is commissioned to drive towards a given destination. As it starts to drive, the robot accelerates until it reaches its specific maximum velocity. It drives further with this velocity until it reaches its destination or until it detects a collision danger and it is forced to brake. Once the collision danger has passed away, the robot resumes its drive towards its destination. In order to determine collision danger, the robot computes its collision distance with respect to the visible obstacles in its environment. To review the details of the collision distance computation the reader is directed to Section 3.1 in Chapter 3 of this thesis.

4.3. Requirements Elicitation and Analysis

This section takes the informal description of the revisited motivational example from Section 4.2 and derives an informal specification and a formal specification of the system requirements, which are presented in Section 4.3.1 and Section 4.3.2 respectively.

4.3.1. Informal Specification of System Requirements

The system requirements in this section are formulated according to the patterns and rules introduced in [JPQ$^+$16]: (1) requirements are always written in the active form, (2) requirements are always written as complete sentences, (3) requirements express processes or activities with the help of process verbs, e.g., accelerate or brake, and (4) exactly one requirement is formulated for each process verb (cf. Chapter 3).

The high-level description of the motivational example given in Section 4.2 serves as a basis for the informal specification of the system requirements in this section. The system in the motivational example is a robot commissioned to drive towards a predetermined destination without actively colliding with any obstacle in its environment. While the behavior of the robot remains unchanged, the informal description provided in the revisited motivational example in Section 4.2 relaxes several constraints describing the allowed and forbidden behavior of the obstacles in the robot's environment. The informal description of the revisited motivational example shows that the behavior of the obstacles in the robot's environment is probabilistic in nature, since an obstacle has a probabilistic choice between several actions that it can execute, e.g., either remain on its current lane or change to another lane. Some of the system requirements introduced in Section 3.3 are therefore reformulated in order to capture the probabilistic nature of the obstacles' behavior in the robot's environment.

A systematic method has been applied to in order to give an overview of the whole catalog of system requirements in the revisited motivational example and to compare the reformulated system requirements with respect to the original requirements (see Figure 4.8). The requirements are displayed in a table form, one requirement in each table row, and identified by their original requirements identification numbers used in Chapter 3. For comparison purposes, the original and the reformulated text of each requirement are displayed next to each other, with the reformulated part being highlighted in bold italic text.

Original Requirement ID	Original Requirement Text	Part of the Original Requirement Text with ***Reformulated Part***

Figure 4.8.: Systematic Method for the Display and Comparison of the Reformulated System Requirements with the Original System Requirements.

Similarly to the requirements elicitation and analysis carried out in Chapter 3, the complete catalog of system requirements for the revisited motivational example is depicted three tables. Table 4.1 shows the allowed and forbidden system actions in the system's environment, while the following two tables describe the system requirements related to the normal operation mode (Table 4.2) and respectively the collision avoidance mode of the mobile service robot (Table 4.3).

The majority of the requirements that underwent reformulation belong to Table 4.1, which, along with the robot's actions, describes also the allowed and forbidden behavior of the obstacles in the robot's environment. Due to the probabilistic nature of the obstacles' behavior, the reformulated requirements specify also a lower or an upper bound for the probability with which the respective behavior is expected to take place. To begin with, the requirements referring to the forwards movement (FR1) the and the remaining at rest of the dynamic obstacles in the robot's environment (FR3) specify the probabilities for two complementary events. Dynamic obstacles are expected to move forwards, which is expressed by the high probability of occurrence, i.e., at least 0.95 (FR1). At the same time, dynamic obstacles become temporary stationary with a probability of occurrence of at most 0.05 (FR3). Furthermore, dynamic obstacles situated in the blind spots of the robot are not allowed to drive in the same direction as the robot, which is described by a low probability of occurrence, i.e., at most 0.0001 (FR6).

The stop behavior of obstacles in the robot's environment is described probabilistically in FR5, which allows dynamic obstacles, once stopped, to resume their movement with a probability of 0.05. The obstacles' behavior with respect to lane changing is formulated in requirements FR2, FR4 and FR7. While FR2 and FR4 address legal lane changes in accordance with the rules depicted in Figure 4.7, FR7 describes illegal lane changes which may be executed by obstacles in the robot's environment. Although jumping over several lanes is illegal (see Figure 4.7), FR7 accounts for the possibility that there may be obstacles in the robot's environment which choose to ignore these rules and specifies a probability of 0.01 execute a jump over several lanes. In turn, legal lane changes are executed according to a normal probability distribution (FR2 and FR4).

The probabilistic behavior of the obstacles in the robot's environment may affect the capability of the robot to reach its destination. There may exist situations in which the robot does not reach its destination, due to the behavior of obstacles in the robot's environment, e.g., an obstacle unexpectedly blocking the path of the robot. This means that there is no absolute certainty that the robot reaches its predefined destination. In order to account for such situations, the functional system requirement FR9 specifies a probability of at least 0.95 for the event that the robot reaches its destination.

Table 4.1.: Requirements for the allowed and forbidden system actions in the system environment: comparison between initial and revisited motivational example.

ID	**Requirement Text in the Initial Motivational Example**	**Requirement Text in the Revisited Motivational Example**
FR1	The robot shall be able to drive forwards, in an environment where dynamic obstacles also move forwards.	The robot shall be able to drive forwards, in an environment where ***dynamic obstacles move forwards with a probability of at least*** 0.95.
FR2	The robot shall be able to change to a safe target lane, in an environment in which dynamic obstacles do not execute lanes changes.	The robot shall be able to change to a safe target lane, in an environment in which ***dynamic obstacles execute legal lane changes according to the uniform probability distribution***.
FR3	The robot shall not be able to drive backwards, in an environment in which dynamic obstacles do not drive backwards.	The robot shall not be able to drive backwards, in an environment in which ***dynamic obstacles become temporary stationary with a probability of at most*** 0.05.
FR4	The robot shall be able to overcome obstacles only by changing to another lane, in an environment in which dynamic obstacles pass by other obstacles by driving through them.	The robot shall be able to overcome obstacles only by changing to another lane, in an environment in which ***dynamic obstacles pass by other obstacles by executing a legal lane change according to the uniform probability distribution***.
FR5	If stopped, the robot shall be able to resume its driving, in an environment in which dynamic obstacles which have become stationary do not resume their movement.	If stopped, the robot shall be able to resume its driving, in an environment in which ***dynamic obstacles which have become stationary resume their movement with a probability of at most*** 0.05.
FR6	The robot shall be able to perceive the space in front of itself up to a specific configurable sensor horizon limit, in an environment in which obstacles situated in the blind spots of the robot do not drive in the same direction as the robot.	The robot shall be able to perceive the space in front of itself up to a specific configurable sensor horizon limit, in an environment in which ***dynamic obstacles situated in the blind spots of the robot drive in the same direction as the robot with a probability of at most 0.0001***.
FR7	The robot shall not be able to jump over lanes, in an environment in which obstacles do not jump over lanes.	The robot shall not be able to jump over lanes, in an environment in which ***dynamic obstacles jump over lanes with a probability of at most*** 0.01.

Table 4.2.: Requirements for the normal operation mode of the robot: comparison between initial and revisited version the motivational example.

ID	Requirement Text in the Initial Motivational Example	Requirement Text in the Revisited Motivational Example
FR8	The robot shall drive towards a given destination.	The robot shall drive towards a given destination.
FR8.1	The robot shall accelerate until it reaches a specific maximum speed, as long as it has not reached its destination and as long as it has not detected any collision danger.	The robot shall accelerate until it reaches a specific maximum speed, as long as it has not reached its destination and as long as it has not detected any collision danger.
FR8.2	The robot shall continue driving with its specific maximum speed, as long as it has not reached its destination and as long as it has not detected any collision danger.	The robot shall continue driving with its specific maximum speed, as long as it has not reached its destination and as long as it has not detected any collision danger.
FR8.3	The robot shall resume driving, if it has stopped due to collision danger and if it has not reached its destination and if it the obstacle has passed by.	The robot shall resume driving, if it has stopped due to collision danger and if it has not reached its destination and if it the obstacle has passed by.
FR8.4	The robot shall resume driving, if it has stopped due to collision danger and if it has not reached its destination and if it detects at least one safe target lane.	The robot shall resume driving, if it has stopped due to collision danger and if it has not reached its destination and if it detects at least one safe target lane.
FR8.5	The robot shall resume accelerating, if it detects at least one safe target lane during its brake maneuver.	The robot shall resume accelerating, if it detects at least one safe target lane during its brake maneuver.
FR9	The robot shall reach the given destination.	The robot shall reach the given destination ***with a probability of at least*** 0.95.
FR9.1	The robot shall start braking, when it approaches its destination.	The robot shall start braking, when it approaches its destination.
FR9.2	The robot shall stop, when it reaches its destination.	The robot shall stop, when it reaches its destination.

Table 4.3.: Requirements for the collision avoidance mode of the robot: a comparison between initial and revisited motivational example.

ID	**Requirement Text in the Initial Motivational Example**	**Requirement Text in the Revisited Motivational Example**
FR10	The robot shall detect collision danger in front of itself in its sensor horizon.	The robot shall detect collision danger in front of itself in its sensor horizon.
FR11	The robot shall detect safe target lanes.	The robot shall detect safe target lanes.
FR12	The robot shall apply collision avoidance measures, if it detects collision danger on the ego lane.	The robot shall apply collision avoidance measures, if it detects collision danger on the ego lane.
FR12.1	The robot shall change to a safe target lane, if it detects collision danger on the ego lane and if it detects a safe target lane.	The robot shall change to a safe target lane, if it detects collision danger on the ego lane and if it detects a safe target lane.
FR12.2	The robot shall brake, if it detects collision danger on the ego lane and if it does not detect any safe target lane.	The robot shall brake, if it detects collision danger on the ego lane and if it does not detect any safe target lane.
FR13	The robot shall check if further drive with reduced speed is possible.	The robot shall check if further drive with reduced speed is possible.
FR13.1	The robot shall consider that further drive with reduced speed is possible if it detects no collision danger on the ego lane.	The robot shall consider that further drive with reduced speed is possible if it detects no collision danger on the ego lane.
FR13.2	The robot shall reduce its speed and maintain it, if further drive with reduced speed is possible.	The robot shall reduce its speed and maintain it, if further drive with reduced speed is possible.
FR13.3	The robot shall brake until it stops, if further drive with reduced speed is not possible.	The robot shall brake until it stops, if further drive with reduced speed is not possible.
FR14	The robot shall remains at rest, if it has stopped due to collision danger and as long as it has not detected a safe target lane and as long as the obstacle has not passed by.	The robot shall remains at rest, if it has stopped due to collision danger and as long as it has not detected a safe target lane and as long as the obstacle has not passed by.

4.3.2. Formal Specification of System Requirements

Due to the inherent uncertainty which characterizes the operational environment of an autonomous safety-critical system, the concept of absolute correctness of a safety-critical system with respect to its system requirements is replaced by upper and/or lower bounds on the probability that desired system behavior may occur. This further leads to the realisation that system requirements of autonomous safety-critical systems are in fact probabilistic in nature.

In the spirit of the specification pattern system proposed by Dwyer et al. in [DAC98, DAC99], Grunske develops the specification pattern system ProProST which allows system designers to specify probabilistic properties for a system under analysis. Each pattern describes a generalized recurring property, i.e., probabilistic existence, and provides a formal specification template in continuous stochastic logic (CSL) (cf. [Gru08]), which is an extension of PCTL for CTMCs (cf. [ASSB96], [BKH99]). Furthermore, the formal specification templates provided in the ProProST pattern system can also be formulated in PCTL and in PCTL* for discrete-time models (cf. [Gru08]). Another formal language which allows the formulation of probabilistic requirements is the STIMULUS language [JG16]. The STIMULUS language enables the description of probabilistic requirements in a manner which bridges the gap to the textual requirements specification. Toennemann uses this capability of the STIMULUS language in order to define a scenario-based test generation strategy, which introduces randomization into the testing process and defines the probabilities with which certain events are expected to take place (cf. [Toe20]).

The definition of the concept of collision danger plays a central role in the formal specification of the system requirements. In Chapter 3, the concept of collision danger is predicated on the environment constraint which prohibits dynamic obstacles in the robot's environment from performing lane changes. The relaxation of this constraint in the revisited motivational example justifies a reworking of the collision danger concept, as shown in Definition 4.3.1.

Definition 4.3.1 - Collision Danger
Let $n \in \mathbb{N}_{>0}$ be the number of obstacles in the environment of an autonomous mobile robot. An obstacle $O_i, i \in [1..n]$ is considered to be a *collision danger* for the robot if and only if the following two statements are true simultaneously:

1. the obstacle O_i is a visible obstacle, and
2. the obstacle O_i is situated inside the collision distance of the robot.

■

Notice that the ability of the dynamic obstacles to carry out lane changes means that, for the mobile service robot, the collision danger may come not only from obstacles situated on the ego lane, but also from obstacles which move on other lanes. The corresponding first-order logic predicate is depicted in Equation (4.26). Notice that the revisited predicate for collision danger is formulated with the help of the non-parametric predicate for visible obstacles, which has been introduced in Definition 3.3.4 and Equation (3.9) in Chapter 3.

$$\begin{aligned} \mathit{CollisionDanger} :& \mathit{VisibleObstacle} \wedge \\ & (0 \leq x_O - x_R) \ \wedge \ (x_O - x_R \leq d^R_{\mathit{Collision}}) \end{aligned} \tag{4.26}$$

Liveness Properties

The system requirement FR9 is part of the functional system requirements, which describe the normal operation mode of the system in Table 4.2, and states that the robot shall reach the given destination. Such a requirement is usually formalized as a liveness property, which asserts that “something good eventually happens” (cf. Chapter 3). Nevertheless, this requirement is also probabilistic in nature due the uncertainty in the robot’s environment caused by the probabilistic obstacle behavior. A probabilistic *liveness property* specifies a lower bound on the probability with which a specific condition is supposed to hold or an event is expected to happen. Informally, the probabilistic liveness property corresponding to requirement FR9 can be formulated as follows:

Probabilistic Liveness Property (Informal Specification). The robot shall eventually reach its destination with a probability of at least 0.95.

The probabilistic computational tree logic (PCTL) is chosen for the specification of this liveness property. In PCTL [HJ94, CG04], liveness can be expressed with one of the state formulae $P_{\geq p}(F\psi)$ or $P_{>p}(F\psi)$ (cf. [KSZ14]). The PCTL state formula $P_{\geq p}(F\psi)$ requires that, under all schedulers, the event represented by the path formula $F\psi$ occurs with at least probability p (cf. [BdAFK18]). The operator F is the existential quantifier which asserts that a state or a set of states of the system model which satisfy the property ψ is eventually reached. In the case of the robot’s liveness property, ψ is the predicate *destinationReached*, as illustrated in Equation (4.27).

$$\theta : P_{\geq 0.95} \ (F \ \mathit{destinationReached}) \tag{4.27}$$

Notice that the liveness property is very similar to the probabilistic existence pattern identified by Grunske [Gru08], which states that a given state or event will hold eventually within a specific time bound. However, in Equation (4.27), no time bound is specified. In the example of the mobile service robot, this means that, with a probability of at least 0.95, the robot will eventually reach its destination, without placing any constraint on how much time it takes for the robot to arrive at its destination.

Notice that there is no relation of equivalence between the PCTL liveness property in Equation (4.27) and the CTL liveness property introduced in Equation (3.12) in Chapter 3. As a matter of fact, there is no qualitative PCTL formula which is equivalent to $AF\psi$ (cf. [BK08]). In general, the PCTL formula $P_{>0}(F \ \psi)$ is equivalent to the CTL formula $EF \ \psi$ (cf. [BK08]).

4.4. Safety Analysis

This section carries out the safety analysis using the HARA method based on the high-level description of the system functionality in the revisited motivational example presented in Section 4.2 and the system requirements described in Section 4.3. Ensuring absolute safety for autonomous safety-critical systems in dynamic and uncertain environment is not feasibly possible and often manufacturers try to build a “safe enough” system for their costumers. Therefore, the HARA analysis carried out in Section 4.4.1 is followed by an analysis in Section 4.4.2 of what “safe enough” means for users of autonomous safety-critical systems. As a result of the HARA analysis, the system safety requirements are extended with the respective environment assumptions in Section 4.4.3. Their informal specification is shown in Section 4.4.4, while the respective formal specification is depicted in Section 4.4.5.

4.4.1. HARA Analysis of the Revisited Motivational Example

For the hazard analysis carried out in this section, the same aspects are considered as in Section 3.4, with the equivalence classes and parameter values corresponding to the revisited motivational example. The aspects considered for the safety analysis in this section are:

- the *location* where the mobile robot operates:

$$\mathit{Location} := \{\mathit{Indoors}, \mathit{Outdoors}\}$$

- the *geometry* of the physical world in which the robot moves, specified through three parameters, namely the number of lanes $n_{\mathit{Lanes}} \in \mathbb{N}_{>0}$, the ground inclination $\alpha_{\mathit{Ground}} \in \mathbb{R}_{>0}$, and the curvature of the road $\gamma_{\mathit{Road}} \in \mathbb{R}_{\geq 0}$:

$$\mathit{PhysicalWorldGeometry} := \{(n_{\mathit{Lanes}} \geq 3,\ \alpha_{\mathit{Ground}} = 0°, \gamma_{\mathit{Road}} = 0°), \dots\}$$

- the *driving conditions*:

$$\mathit{DrivingConditions} := \{(\mathit{Dry}, \mathit{Non\text{-}slippery}), (\mathit{Wet}, \mathit{Slippery}), \dots\}$$

- the *environment* in the sensor horizon of the robot:

$$\mathit{Environment} := \{\mathit{DynamicObst}, \mathit{StationaryObst}, \mathit{TemporaryStationaryObst}\}$$

- the *system usage*, such as driving towards a predefined destination,
- the *system behavior*:

$$\begin{aligned}
\mathit{SystemBehavior} &:= \mathit{RobotLaneChangingBehavior} \times \mathit{RobotDrivingBehavior}\\
\mathit{RobotDrivingBehavior} &:= \{\mathit{Accelerating}, \mathit{FullSpeedDrv}, \mathit{ReducedSpeedDrv},\\
&\quad \mathit{Braking}, \mathit{Stopped}\}\\
\mathit{RobotLaneChangingBehavior} &:= \{\mathit{LeftLaneChange}, \mathit{RightLaneChange}, \mathit{LaneKeep}\}
\end{aligned}$$

- the *environment behavior*, which refers to the behavior of the obstacles situated in the robot's sensor horizon:

$$\begin{aligned}
&\mathit{EnvironmentBehavior} && := \mathit{ObstLaneChangingBehavior} \times \mathit{ObstDrivingBehavior}\\
&\mathit{ObstLaneChangingBehavior} && := \mathit{ObstDepartLane} \times \mathit{ObstAction} \times \mathit{ObstArrivalLane}\\
&\mathit{ObstAction} && := \{\mathit{LeftLaneChange}, \mathit{RightLaneChange}, \mathit{LaneKeep},\\
& && \mathit{LaneJump}\}\\
&\mathit{ObstDrivingBehavior} && := \mathit{ObstSpeed} \times \mathit{ObstDirection} \times \mathit{ObstLocation}\\
&\mathit{ObstSpeed} && := \{\mathit{FullSpeed}, \mathit{HighSpeed}, \mathit{LowSpeed}, \mathit{Stopped}\}\\
&\mathit{ObstDirection} && := \{\mathit{Opposing}, \mathit{Following}, \mathit{Leading}, \mathit{RunningAway}\}\\
&\mathit{ObstLocation} && := \mathit{ObstPosition} \times \mathit{ObstLane}\\
&\mathit{ObstPosition} && := \{\mathit{InFront}, \mathit{Behind}\}
\end{aligned}$$

  where *ObstDepartLane*, *ObstArrivalLane*, $\mathit{ObstLane} \in \Omega_{\mathit{Lane}}$ and *Stopped*, *FullSpeed*, *HighSpeed*, *LowSpeed*, $\subset \Omega^{O_i}_{\mathit{Vel}}$.

The environment behavior consists of a combination of the lane changing behavior and the driving behavior of the obstacles in the robot's environment. This combination is modeled as the Cartesian product between the set of events related to the lane changing behavior and the set of events describing the obstacles driving behavior. The events modeling the lane changing behavior of a visible obstacle specify the obstacle's depart and arrival lane along with the respective action carried out by the obstacle. Notice that the depart lane and the arrival lane are elements of the sample space Ω_{Lane}. For the sake of simplicity, the events modeling the obstacle lane changing behavior are referred hereafter only by the action executed by the obstacle. Thus, the lane changing behavior with respect to a visible obstacle consists of the following events: (1) *LaneKeep* - obstacle stays on its current lane, (2) *LeftLaneChange* - obstacle makes a lane change to the lane on its left, (3) *RightLaneChange* - obstacle makes a lane change to the lane on its right, and (4) *ObstLaneJump* - obstacle jumps over one or several lanes.

The driving behavior in the robot's environment is in turn modeled by events which describe the speed, the direction of movement, and the location of the obstacle. The events which describe the obstacle speed are: (1) *FullSpeed* - obstacle drives at full speed, (2) *HighSpeed* - obstacle drives with high speed, (3) *LowSpeed* - obstacle drives with low speed, and (4) *Stopped* - obstacle is stopped. Notice that the events *FullSpeed*, *HighSpeed*, *LowSpeed*, and *Stopped* are elements of the sample space $\Omega^{O_i}_{\mathit{Vel}}$ specific to the velocity changes of obstacle $O_i, i \in \mathbb{N}_{\geq 1}$, visible in the robot's environment. The position of the obstacle is expressed in relation of the robot. Thus, the obstacle can be situated in front of the robot (*InFront*) or behind the robot (*Behind*). The movement direction of the obstacle is also considered in relation of the robot. In general, for a moving physical object the notion of movement direction is strongly related with the concept of orientation. For an autonomous robot in particular the orientation of a robot is also called bearing or heading direction (cf. [TBF05]). The orientation of the robot described by an angle θ together with its position expressed in Cartesian coordinates (x_R, y_R) makes the robot's pose with respect to the Cartesian coordinate system (x, y).

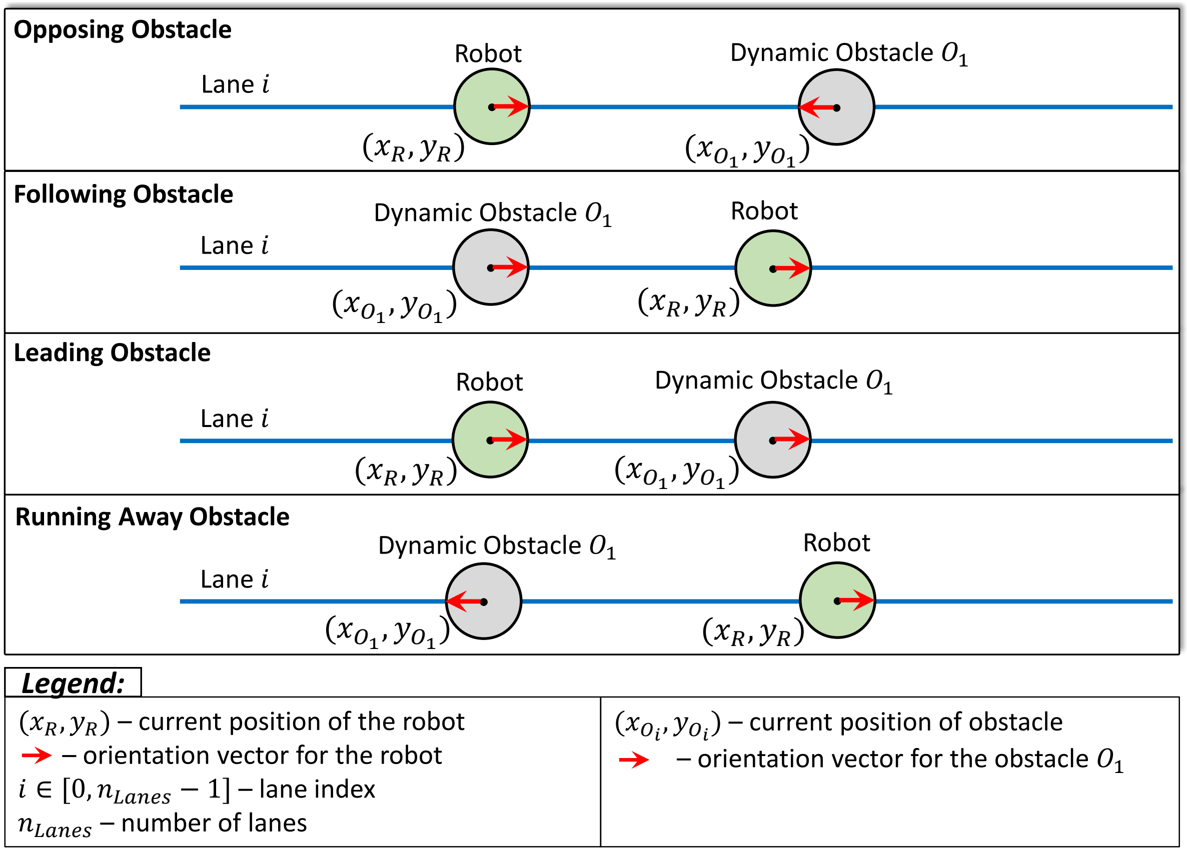


Figure 4.9.: Visual Intuition of the Movement Directions of Dynamic Obstacles with respect to the Mobile Service Robot.

The requirement FR6 prohibits dynamic obstacles situated in the blind spots of the robot to move in the same direction as the robot, as this can lead to a collision actively caused by the robot. In Figure 4.6 these are areas situated in front of the robot beyond its sensor horizon, areas of the neighboring lanes not covered by the robot's sensors, and areas behind the robot outside its safety net. In order to detect collision dangers and avoid possible collisions, the robot must be able to recognize the direction of movement of the dynamic obstacles in its environment relative to itself. The HARA analysis carried out in this section defines specific events that can be used to describe the direction of movement for dynamic obstacle in relation of the robot: (1) *Opposing* - the obstacle drives in front of the robot in the opposing direction of movement, (2) *Following* - the obstacle drives behind the robot in the same direction of movement, following the robot, (3) *Leading* - the obstacle drives in front of the robot in the same direction of movement, leading the robot, and (4) *RunningAway* - the obstacle drives behind the robot in the opposing direction of movement, distancing itself from the robot. Figure 4.9 gives a visual intuition of the events defined for the movement direction of dynamic obstacles. The movement direction of dynamic obstacles is described from the robot's point of view. This dimension is orthogonal to the one expressed in the requirements FR1 and FR3 in Table 4.1, which talk about the forwards movement and the backwards movement of dynamic obstacles in the robot's environment from the obstacle's point of view.

The values of the defined parameters are combined with each other in order to create unique concrete situations, with one situation being depicted in one table row of Table 4.4. There is an uncountable number of ways in which the obstacles in the robot's environment may behave. Thus, neither the list of identified situations nor the list of selected parameters are considered to be complete. Furthermore, the presence of some parameters depends on other parameters. Depending on the location of the robot's environment, it makes sense to consider driving conditions, for example in case of an outdoors environment, or to leave them out of the HARA analysis, like in case of an indoors environment.

The HARA analysis defines for every identified situation a possible hazard and the potential effect which the hazard may have. Throughout the HARA analysis it is considered that the operational environment of the robot is situated indoors, consists of a straight road with three lanes and ground inclination of 0°. The environment is populated with dynamic obstacles. The three lanes are identified through the index $i \in \{0, \ldots, n_{Lanes}\}$, where $n_{Lanes} = 3$. Initially, lane 0 is considered to be the ego lane. In all the situations identified in the HARA analysis, the system usage remains unchanged, namely that the robot drives towards a given destination. The situations identified through this analysis can be roughly split into two categories. The first category emphasizes the lane changing behavior of the obstacles in the robot's environment, while the second category highlights their driving behavior.

Take for example the situations H1 depicted in Table 4.4, in which the mobile service robot drives at full speed towards its given destination. Consider there is a dynamic obstacle O_1 moving from the opposite direction on the ego lane (*Opposing*). With probability of 0.9999, obstacle O_1 continues to move on the ego lane (*LaneKeep*), instead of executing a lane change. The hazard is that the robot does not activate any collision avoidance measures in time, e.g. changing to a safe lane. The potential effect of this hazard is that the robot does not brake on time, which may result in a frontal crash with obstacle O_1 actively caused by the robot. A slightly different example is the situation H2, in which the robot accelerates on lane 0 towards its destination. Consider that on lane 1 two dynamic obstacles O_2 and O_3 move from the opposite direction (*Opposing*), with obstacle O_3 situated behind obstacle O_2. With probability of 0.9999, obstacle O_3 executes a legal lane change to lane 0 (*RightLaneChange*) on the ego lane, in an attempt to pass by the obstacle O_2. The hazard in this situation is that the robot has very little to no reaction time and therefore does not deploy any collision avoidance maneuvers, e.g. braking. The potential effect is a frontal or side crash with the obstacle O_3 actively caused by the robot. Notice that situations H1 and H2 are two examples of the first category of situations which focuses on the lane changing behavior of the obstacles in the robot's environment. At the same time, both situations exemplify legal moves with respect to the lane changing behavior - O_1 maintaining its lane in H1 and O_3 making a lane change from lane 1 to lane 0 in H2.

Table 4.4.: Mobile Service Robot - Hazard Analysis and Risk Assessment in the Revisited Motivational Example.

ID	**Location**	**Phys. World Geometry**	**Environment**	**System Usage**	**System Behavior**	**Environment Behavior**	**Hazard**	**Potential Effect**
H1	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground} = 0°$, $\gamma_{Road} = 0°$	*DynamicObst*	*Driving towards a given destination*	*FullSpeedDrv*	*Opposing* movement, *LaneKeep* on the ego lane with $P \geq 0.9999$	No robot's brake	Frontal crash
H2	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground} = 0°$, $\gamma_{Road} = 0°$	*DynamicObst*	*Driving towards a given destination*	*Accelerating*	*Opposing* movement, *RightLaneChange* to the ego lane with $P \geq 0.9999$	No robot's brake	Frontal/ Side crash
H3	*Indoors*	$n_{lanes} \geq 3$, $\alpha_{Ground} = 0°$, $\gamma_{Road} = 0°$	*Temporary Stationary Obst*	*Driving towards a given destination*	*FullSpeedDrv*	*Opposing* movement, *LowSpeed* on the ego lane with $P \geq 0.5$	No robot's stop	Frontal crash
H4	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground} = 0°$, $\gamma_{Road} = 0°$	*DynamicObst*	*Driving towards a given destination*	*Braking*	*Following* movement in blind spot on the ego lane with $P \geq 0.001$	No robot's stop	Rear crash
H5	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground} = 0°$, $\gamma_{Road} = 0°$	*DynamicObst*	*Driving towards a given destination*	*FullSpeedDrv*	*Opposing* movement, *LaneJump* on the ego lane with $P \geq 0.1$	No robot's brake	Side crash
H6	*Indoors*	$n_{Lanes} \geq 3$, $\alpha_{Ground} = 0°$, $\gamma_{Road} = 0°$	*DynamicObst*	*Driving towards a given destination*	*FullSpeedDrv*	*Opposing* movement, *FullSpeed* on the ego lane with $P \geq 0.9999$	No robot's brake	Frontal crash
...	...	...	...	...	...	...	...	...

A third example of lane changing behavior is the one depicted in situation H5. The robot drives on lane 0 at full speed towards its destination. Consider that on lane 2 a dynamic obstacle O_4 moves from the opposite direction (*Opposing*). With probability of 0.1, obstacle O_4 makes a lane jump from lane 2 to lane 0 (*LaneJump*). In such a situation, the hazard is that the robot does not activate any collision avoidance maneuvers, e.g. braking, due to very little to no reaction time. The potential effect of this hazard is a side crash with obstacle O_4 actively caused by the robot. In contrast to the situations H1 and H2, the situation in H5 represents an illegal lane change.

Three examples of the driving behavior of the obstacles in the robot's environment are depicted in situations H3, H6 and respectively H4 of Table 4.4. In the situations H3 and H6 the robot drives at full speed towards its predefined destination. In H3, the temporary stationary obstacle O_5 starts to move with probability of 0.5 (*LowSpeed*) on the ego lane in the opposite direction with respect to the robot (*Opposing*). In H6, a dynamic obstacle O_6 drives at full speed on the ego lane with probability of 0.9999 (*FullSpeed*) in the opposite direction of the robot (*Opposing*). In both situations, the hazard is that the robot cannot brake to a full stop, which may have as a consequence a frontal crash with the obstacles O_5 and respectively O_6 actively caused by the robot. In H4, the robot is braking, while a dynamic obstacle O_7 situated in a blind spot of the robot is following it with a probability of at least 0.001. The hazard is that the robot cannot finish the braking process and cannot come to a full stop, which may have as a potential effect a rear crash with obstacle O_7 caused by the robot.

4.4.2. "Safe Enough" for Autonomous Safety-Critical Systems

The HARA analysis carried out on the example of the mobile service robot has identified several hazards. For each of these hazards, the potential effect is a collision with an obstacle in the robot's environment actively caused by the robot. The collision is a safety-critical event and potentially causes harm or damages both to the robot as well as to the obstacles in its environment. In order to avoid such hazards and their effects on the system and its environment, the system designers define safety requirements which are used during the system development process to verify and validate the system.

The safety of autonomous safety-critical systems which operate in uncertain, dynamic environments cannot be ensured on an absolute basis. Instead there is always some residual risk, which remains after all implemented safety measures have been deployed (cf. [Int97b], [Int11b]). This type of risk appears due to the uncertainties present in the system's operational environment and is not covered by the implemented safety measures. How much residual risk is acceptable for autonomous safety-critical systems is a question which has been researched by various authors. Blumenthal et al. [BFBBI] examine different approaches for assessing whether autonomous vehicles are acceptably safe. These approaches have been gathered through interviews with different stakeholders in the area of autonomous driving, with a survey of the general public and through review of literature. The report categorizes the identified approaches in three different categories: safety as a measurement, safety as a process, and safety as threshold. *Safety as a measurement* is defined as a quantitative measure for the safety performance of autonomous vehicles using data-driven evidence and it can be indicated by a leading measure, i.e. measures of pre-crash driving behavior, or by a lagging measure, i.e. measures of collisions and post-collision outcomes (cf. [BFBBI]). *Safety as a process* indicates the developer behavior necessary for the achievement of safety, which encompasses the engineering efforts in system-level verification and validation that together with other practices, e.g. technical standards, compliance with goverment regulations and safety culture, contribute to a safety case (cf. [BFBBI]). *Safety as a threshold* can be defined as a goal based on human driving performance, or as a goal based on the technological potential of the autonomous driving system (cf. [BFBBI]).

Liu et al. [LYX19] set to find out what safety as threshold predicated on the driving performance of the human driver is and ask the question *How safe is safe enough?* with respect to self-driving vehicles in a survey conducted between October and November 2016 in China. Liu et al. [LYX19] introduce two quantitative metrics for risk measurements: *risk frequency* which describes the likelihood of the occurrence of traffic crashes with a specific risk severity, and *risk-acceptance rate* which is defined as the percentage of the respondents which are ready to accept traffic scenarios, given specific risk frequency and risk severity. Both risk metrics introduced by Liu et al. are expressed at two different levels: as one fatality per certain population number and as one fatality per vehicle-kilometers traveled. Through interviews with the survey subjects, the authors determined the risk-acceptance rates for traffic scenarios with varying risk frequencies in human-driven vehicles and in self-driving vehicles. Based on the given risk-acceptance rates the acceptable risk frequencies were predicted for both human-driven vehicles as well as for self-driving vehicles. The predicted risk frequencies for human-driven vehicles were compared with those predicted for self-driving vehicles, thus giving the *tolerable risk* criterion, while the predicted risk frequencies of self-driving vehicles were compared against the current global road traffic risk, which gives the *broadly acceptable risk* criterion. With respect to the *tolerable risk* criterion, the results of the study carried out by Liu et al. [LYX19] show that autonomous driving vehicles should be four or five times as safe as human drivers, meaning that the usage of autonomous vehicles should reduce $75\% - 80\%$ of current driver-related traffic fatalities in order to be accepted by the general public (cf. [LYX19]). With respect to the *broadly acceptable risk* criterion, the authors found out that that the predicted risk frequencies in self-driving vehicles needs to be two orders of magnitude lower than the current global road traffic risk (cf. [LYX19]).

Waycaster et al. [WMB$^+$18] carried out a study with respect to new regulations and other reactions, e.g. recalls, to fatal accidents in different modes of transportation implemented between 2002 and 2009 (cf. [WMB$^+$18]). In addition, the authors study two major fatal accident investigations from commercial aviation and two major automotive recalls associated with fatal accidents and estimate the cost per expected fatality prevented for these reactions. The authors suggest that individuals increase their demand for safety if an automated system is in charge.

In a study to replicate the "better-than-average" effect, Nees [Nee19] interviewed a sample of US drivers and asked them to rate (1) their own perceived ability to drive safely, (2) the ability of current vehicles with automation to drive safely and (3) their desired level of safety before they would accept self-driving cars. The author took into consideration three outcomes of the desired safety of the participants, namely the desired safety level of self-driving cars (1) before such vehicles are allowed on public roads, (2) before the participants would feel reasonably safe riding in such vehicles, and (3) before the participants themselves would buy a self-driving vehicle, all other things being equal (cf. [Nee19]). The results show that 80.9% of the interviewed subjects consider themselves more safe than average (higher that the 50th percentile). Furthermore, 66.1% of the sample rated themselves as more safe than the current self-driving cars, while 68, 1% desired self-driving cars to be safer than they perceives themselves to be (cf. [Nee19]). Nevertheless, according to the results of the study, the modal participant expressed that they desired a level of safety in the 95th up to the 99th percentile. The author suggests that the participants may not have been using their own perceived safety percentile as a mean for comparison but instead wished that self-driving car be as safe as the best human drivers (cf. [Nee19]).

4.4.3. Extending Safety Requirements with Environment Assumptions

The safety analysis carried out in this section identified various hazards which have as potential effect the collision of the mobile service robot with one of the obstacles which populate its environment. These hazards can occur at runtime due to the obstacle behavior which does not correspond anymore to the environment assumptions defined and used by the system developers in the system's design phase. To guard against the causes which lead to the identified hazards, the environment assumptions are captured explicitly in the safety requirements specification of the system.

The HARA analysis identifies two categories of situations that can lead to hazards with safety-critical potential effects. The first category relates to the lane changing behavior of the obstacles in the robot's environment, while the second category takes into consideration the driving behavior of the obstacles. It follows that the environment assumptions formulated to guard against such situations will refer on one side to the lane changing behavior of the obstacles and on the other side to their driving behavior. Notice that there does not exist a 1-to-1 relation between the hazardous event and the environment assumption that covers it. In fact, one environment assumption can correspond to more than one situations that result in hazards with safety-critical effects. Figure 4.10 depicts a requirements pattern which is applied in order to extend the safety requirements specification of the mobile service robot with the environment assumptions.

The pattern proposed in Figure 4.10 is inspired by the work of Rupp and Die Sophisten [JPQ+16]. Similar patterns have been proposed by Mauritz [Mau19] and Toennemann [Toe20]. The requirements pattern proposed by Mauritz is used to formulate the system requirements for a lane change assistant (cf. [Mau19]). In turn, the pattern presented by Toennemann is used to formulate the functional system specification of an automotive function, which supervises the correct application of voltage to the motor based on the target current and on the measured current (cf. [Toe20]).

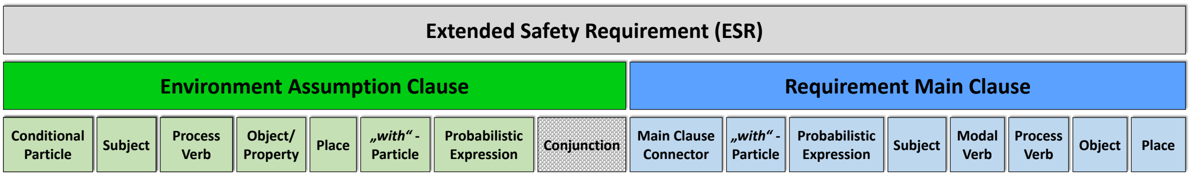


Figure 4.10.: Pattern for the Safety Requirements Specification extended with Environment Assumptions.

The extended safety requirement consists of two parts: an environment assumptions clause and a requirement main clause. The environment assumption clause contains the following parts: (1) conditional particle, (2) subject, (3) process verb, (4) object, (6) probabilistic expression preceded by (5) the *with*-particle, and optionally (7) a conjunction particle. The *conditional particle* introduces a logical statement about an event or an action taking place in the system's environment. The action is described with the help of the *process verb* and is executed by the *subject*. The action carried out by the subject can change a *property* of the subject or can affect an object. The *object* represents an item or an actor on which the effects of the action executed by the subject are reflected. Optionally, the *place* can be specified where the subject of the environment assumption executes the action specified in the process verb. The *probabilistic expression* preceded by the *with*-particle specifies the probability with which the subject is assumed to perform the action described by the process verb. The *conjunction* particle allows to combine several assumptions through logical conjunction. Some environment assumptions may refer to the same aspect of the environment and can be combined with each other through the conjunction particle, while other environment assumptions work better if they are expressed separately. Therefore, it is left to discretion of the system requirements engineers to appreciate and decide whether the usage of the conjunction particle is appropriate or not.

The main clause of the safety requirement is built in a similar manner to the environment assumption and consists of the following parts: (1) main clause connector, (3) probabilistic expression, preceded by (2) the *with*-particle, (4) subject, (5) modal verb, (6) process verb and (7) object. The *main clause connector* is a particle which connects the requirement main clause to the environment assumption. The *probabilistic expression* specifies the probability with which the *subject* executes a certain action, described by the *process verb*. The *modal verb* describes different legal meanings which a requirement may carry for different system stakeholders, and can be one of the three verbs: *shall*, *should*, or *will*. (cf. Chapter 3). The modal verb is followed directly by the *process verb* in the main clause, unless the action expressed by the process verb is negated. In this case, the negation particle *not* is interposed between the modal verb and the process verb and is considered to be part of the modal verb construction. The negation is used when the process verb specifies an action which the subject is forbidden to carry out, e.g. *shall not accelerate*. The *object* represents an item or an actor which is affected by the action executed by the subject. Notice that the object of the requirement main clause can be the subject of the environment assumption, e.g. an obstacle in the robot's environment. Optionally, the *place* can be specified where the action described through the process verb is executed by the subject.

4.4.4. Informal Specification of Extended Safety Requirements

The potential effect of the hazards identified through the HARA analysis is an active collision of the robot with an obstacle in its environment. In order to guard against the effect of these hazards on the system and its environment, system designers define safety requirements, which are used in the system development process to verify and validate the behavior of the system in its environment.

A safety requirement can be formulated as formal safety property, since the intention of a safety requirement is to prevent a specific hazardous event from ever happening. At a general level, a *safety property* informally states that "nothing bad ever happens". In order to meet the desired level of safety expressed in user interviews [Nee19], the safety property must specify a quantitative level for the probability with which the safety property holds. For the purpose of this example, the robot's safety property derived from the results of the HARA analysis is transcribed into the following informal specification in natural language:

Probabilistic Safety Property (Informal Specification). With a probability of at least 0.99, the robot shall not collide actively with any obstacle in its environment.

Environment assumptions are captured explicitly and extend the safety requirements specification of an autonomous safety-critical system in order to protect the system against the environment behavior which can lead to the safety hazards identified through the HARA analysis. For the example of the mobile service robot, the analysis identifies two types of events that can lead to hazards with safety-critical potential effects. The first type of events is triggered by the lane changing behavior of the obstacles while the second type of events relates to their driving behavior. The environment assumptions can belong to one type or the other depending on what type of hazardous events they cover. The extension of the safety requirements with the environment assumptions is carried out according to the requirements pattern introduced in Section 4.4.3.

The safety requirement of the mobile service robot extended with various environment assumptions is shown in Table 4.5. The last column of the table also shows which of the hazards identified in the HARA analysis are covered by the respective extended safety requirements. Notice that while the probabilistic expression in the requirement main clause specifies a lower bound or an upper bound for the probability with which the subject carries out a certain action, the probabilistic expression in the environment assumption can also specify the probability distribution with which the subject of the environment assumption is assumed to execute specific actions.

Table 4.5.: Mobile Service Robot - Safety Requirement extended with Environment Assumptions.

ID	**Environment Assumption Clause**	**Requirement Main Clause**	**Covered Hazards**
ESR1	If dynamic obstacles in the robot's sensor horizon execute legal lane changes with the uniform probability distribution,	then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	H1 and H2
ESR2	If dynamic obstacles in the robot's sensor horizon execute illegal lane changes with a probability of at most 0.01,	then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	H5
ESR3	If dynamic obstacles follow the robot in a blind spot of the robot sensors with probability of at most 0.0001,	then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	H4
ESR4	If dynamic obstacles in the robot's sensor horizon change their velocity during their drive with the histogram distribution $(0, 0.125, 1, 0.75, 2, 0.125, 3)$,	then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	H6
ESR5	If temporary stationary obstacles start moving in the robot's sensor horizon with a probability of at most 0.05,	then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	H3

4.4.5. Formal Specification of Extended Safety Requirements

To formalize the extended safety requirements, several domain specific concepts which appear in these requirements must be translated into first-order logic predicates. These concepts describe occurrences in the robot's environment in which obstacles are involved: active collision, legal and illegal lane change executed by an obstacle in the robot's environment, obstacle following the robot, and stationary obstacle starting to move. The first concept to be defined is that of active collision introduced in Definition 4.4.1.

Definition 4.4.1 - Active Collision
Let $n \in \mathbb{N}_{\geq 1}$ be the number of visible obstacles in the environment of an autonomous mobile robot. Then it is said that an *active collision* is caused by the mobile robot if and only if the following two conditions hold simultaneously.

1. the robot is not stationary and
2. there exists an obstacle O_i, $i \in \{1, \dots, n\}$, which is situated on the ego lane and inside the robot's safety net.

■

The corresponding first-order logic predicate is given in Equation (4.28). Notice that the predicate in Equation (4.28) is the negation of the bracket expression in the equation of the safety property in Section 3.4.

$$\begin{aligned} \mathit{ActiveCollision}:\ \exists O_i.\ &\neg(v_R == 0) \wedge ((y_R == y_{O_i}) \wedge \\ &(-c_R \leq x_{O_i} - x_R) \wedge (x_{O_i} - x_R \leq c_R)) \end{aligned} \tag{4.28}$$

The next two concepts to be defined are those of legal lane change and respectively illegal lane change executed by an obstacle in the robot's environment.

Definition 4.4.2 - Legal Lane Change
Let $n \in \mathbb{N}_{\geq 1}$ be the number of visible obstacles in the environment of an autonomous mobile robot.Then it is said that *legal lane change* occurs in the mobile robot's environment if and only if for each visible dynamic obstacle O_i, $i \in \{1, \dots, n\}$, the index difference between the two lanes occupied by the obstacle at two consecutive steps t and $t+1$ is equal to 1. ■

The corresponding predicate is shown in Equation (4.29).

$$\mathit{LegalLaneChange}: \forall O_i.\ \neg(y_{O_i}(t) - y_{O_i}(t+1) == 0) \wedge (|y_{O_i}(t) - y_{O_i}(t+1)| == 1) \tag{4.29}$$

Definition 4.4.3 - Illegal Lane Change
Let $n \in \mathbb{N}_{\geq 1}$ be the number of visible obstacles in the environment of an autonomous mobile robot. Then it is said that *illegal lane change* occurs in the mobile robot's environment if and only if there exists a dynamic obstacle O_i, $i \in \{1, \dots, n\}$, for which the index difference between the two lanes occupied by the obstacle at two consecutive steps t and $t+1$ is larger than 1. ■

The corresponding predicate is shown in Equation (4.30).

$$\mathit{IllegalLaneChange} : \exists O_i.\ |y_{O_i}(t) - y_{O_i}(t+1)| > 1 \tag{4.30}$$

A visual intuition for an obstacle following the robot has already been introduced in Figure 4.9. The robot and the dynamic obstacle O_1 are represented in the Cartesian coordinate system in Figure 4.11. Notice that the respective orientation angles of the robot and of obstacle O_1 are $\theta_R = 0°$ and $\theta_{O_2} = 0°$, which means that both have the same direction of movement.

The representation of the robot and the obstacle in Figure 4.11 is consistent with the HARA analysis carried out in Section 4.4.1, in which the environment is considered to be a straight road. This can simplify the analysis considerably as only the orientation angles 0° and 180° must be considered. For the case when obstacles change their lane, the direction vector of the obstacle is decomposed into its respective x-component and y-component and only the x-component is considered in the further analysis. Since the environment of the mobile service robot is a straight road, a simpler method can be used to determine whether an obstacle moves in the same direction as the robot on the x axis or not. Instead of using the angle θ_{O_i} to express the orientation of obstacle O_i, the difference between two consecutive positions on the lane of obstacle O_i and the algebraic sign of this difference can be used to show whether obstacle O_i moves in the same direction as the robot or not. This decision will help the design-time verification, since any additional variable introduced to model a specific feature of the system or of its environment leads to an increase in the state space of the overall system model. Definition 4.4.4 introduces the concept of movement in the same direction for the robot and an arbitrary obstacle $O_i, i \in \mathbb{N}_{\geq 1}$ in the robot's environment.

Definition 4.4.4 - Same Movement Direction
Let $n \in \mathbb{N}_{\geq 1}$ be the number of visible obstacles in the environment of an autonomous mobile robot. Then it is said that dynamic obstacles have the *same movement direction* as the mobile robot if and only if there exists at least one obstacle O_i, $i \in \{1, \ldots, n\}$, for which the difference between two consecutive positions at steps t and $t+1$ has the same algebraic sign as the difference between the robot's consecutive positions at steps t and $t+1$. ■

The corresponding predicate is presented in Equation (4.31):

$$\mathit{SameMovementDirection} :\ \exists O_i.sgn(x_R(t) - x_R(t+1)) == sgn(x_{O_i}(t) - x_{O_i}(t+1)) \tag{4.31}$$

where $sgn : \mathbb{R} \to \{-1, 0, 1\}$ is the signum function with

$$sgn(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

Definition 4.4.5 - Following Obstacle in Robot's Blind Spot
Let $n \in \mathbb{N}_{\geq 1}$ be the number of obstacles in the environment of an autonomous mobile

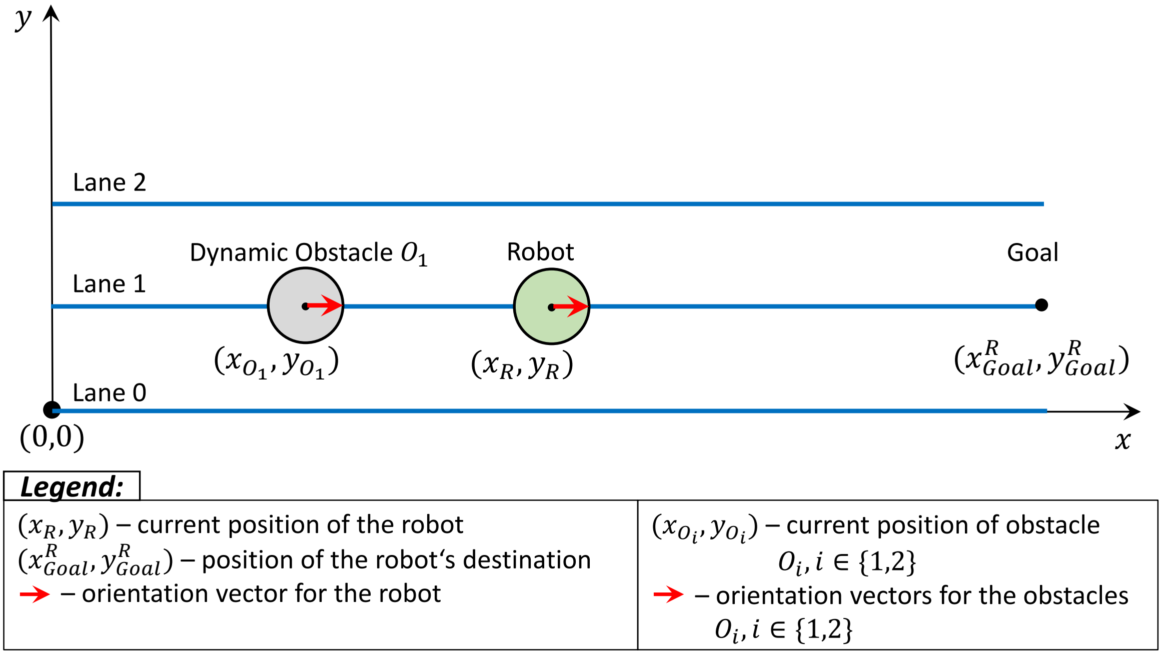

y
Lane 2
Dynamic Obstacle O_1
Robot
Goal
Lane 1
(x_{O_1}, y_{O_1})
(x_R, y_R)
(x^R_{Goal}, y^R_{Goal})
Lane 0
(0,0)
x
Legend:
(x_R, y_R) – current position of the robot
(x^R_{Goal}, y^R_{Goal}) – position of the robot's destination
→ – orientation vector for the robot
(x_{O_i}, y_{O_i}) – current position of obstacle $O_i, i \in \{1,2\}$
→ – orientation vectors for the obstacles $O_i, i \in \{1,2\}$

Figure 4.11.: Representation of a Dynamic Obstacle Following the Robot in the Cartesian Coordinate System.

robot. Then it is said that there is a *following obstacle in a blind spot of the robot* if and only if there exists a dynamic obstacle O_i, $i \in \{1, \ldots, n\}$, which has the same movement direction as the robot and is situated outside the robot's safety net. ■

The corresponding predicate is presented in Equation (4.32):

$$\begin{aligned} FollowingObstacleInBlindSpot : \exists O_i.\ & SameMovementDirection \\ & \wedge x_{O_1}(t) - x_R(t) < -c_R \end{aligned} \tag{4.32}$$

The last concept to be defined is that of obstacle movement (re)start. This concept applies on one side to stationary obstacles which start to move, thus becoming dynamic obstacles and on the other side to dynamic obstacle which have been forced to stop and then resume their movement.

Definition 4.4.6 - Obstacle Movement Restart

Let $n \in \mathbb{N}_{\geq 1}$ be the number of visible obstacles in the environment of an autonomous mobile robot. Then it is said that *obstacles restart their movement* in the robot's environment if and only if there exists at least one obstacle O_i, $i \in \{1, \ldots, n\}$, which is stationary at step t and has a velocity larger that zero at step $t + 1$. ■

The corresponding predicate is found in Equation (4.33):

$$ObstacleMovementRestart :\ \exists O_i.v_{O_i}(t) == 0 \wedge v_{O_i}(t+1) > 0 \tag{4.33}$$

Definition 4.4.7 - Formal Specification of Extended Safety Requirements
Let *ReqMainClause* be the the requirement main clause of an extended safety requirement $ESR\ i$, $i \in \mathbb{N}_{\geq 1}$. Let *ReqProb* be a probability expression that defines the probability with which the main clause of $ESR\ i$ is required to be satisfied. Let *EnvAssumptionClause* be the environment assumption clause of the extended safety requirement $ESR\ i$. Let *EnvAssumptionProb* be a probability expression that specifies the probability with which the environment assumption clause is required to be satisfied. Then the *formal specification of the extended safety requirement ESR i* is given in Equation (4.34):

$$P_{EnvAssumptionProb}(EnvAssumptionClause) \rightarrow P_{ReqProb}(ReqMainClause) \tag{4.34}$$

■

Both the environment assumption clause as well as the requirement main clause are formalized as path formulae which are required to be satisfied with a certain probability specified by the respective probability expression. As already known, P is the probability operator introduced in the logic PCTL. Grammar 4.1 depicts the fragment of the PCTL grammar which is used to describe the path formula. The fragment of PCTL is described in extended Backus-Naur form (EBNF). In this description, X and U denote the temporal operators *Next* and respectively strong *Until* of the PCTL logic. The usual temporal operators, F (*Eventually*), G (*Globally*) can be derived from the strong *Until* operator. The complete PCTL syntax is given in Chapter 2.

$\langle EnvAssumptionClause \rangle$ ::= $\langle PathFormula \rangle$

$\langle ReqMainClause \rangle$ ::= $\langle PathFormula \rangle$

$\langle PathFormula \rangle$ ::= 'X' $\langle StateFormula \rangle$ | $\langle StateFormula \rangle$ 'U' $\langle StateFormula \rangle$

$\langle StateFormula \rangle$ ::= '*true*' | $\langle AtomicProposition \rangle$
| '¬' $\langle StateFormula \rangle$
| $\langle StateFormula \rangle$ '∧' $\langle StateFormula \rangle$

Grammar 4.1.: PCTL Grammar Fragment in EBNF for the Path Formula in the Extended Safety Requirements.

Grammar 4.2 depicts the EBNF grammar used for the formal definition of the probability expression. Grammar 4.2 provides two possibilities to specify the probability expressions of the environment assumption clause and requirement main clause respectively: (1) the probability expression defines a probability bound and (2) the probability expression defines a probability distribution. Notice that the part of the EBNF grammar corresponding to the specification of the probability distribution is inspired by the work of Herold et al. [HKW^{+}08] and the UML Profile for Schedulability, Performance and Reliability [Obj05]. The specification of the probability bound uses the same comparison operators as in the syntax of PCTL introduced in Section 2.3.2. The specification of the probability distribution has been added in order to account for environment assumptions which specify such probability distribution. There are eight probability distributions defined in Grammar 4.2: (1) uniform, (2) binomial, (3) normal, (4) Poisson, (5) gamma, (6) exponential, (7) Bernoulli, and (8) the histogram distribution.

Each probability distribution has its specific parameters. The *uniform* distribution has two parameters $a, b \in \mathbb{R}_{\geq 0}$ that designate the start and the end of the sampling interval. The *binomial* distribution has two parameters, namely the probability of success $p \in [0, 1]$ and a positive integer $n \in \mathbb{Z}_{\geq 0}$ which defines the number of independent experiments. The *normal* or the *Gauss* distribution has two parameters, namely the mean value $\mu \in \mathbb{R}$ and the standard deviation $\sigma \in \mathbb{R}$. The *Poisson* distribution has one parameter, the constant mean rate $\lambda \in \mathbb{R}_{>0}$ with which a given number of events occurs in a fixed interval. The *gamma* distribution has two parameters: a positive integer $k \in \mathbb{Z}_{\geq 0}$ and the mean value $a \in \mathbb{R}_{>0}$, which are given as inputs in the formula of the gamma distribution, $\frac{x^{(k-1)} * e^{-\frac{x}{a}}}{a^k * (k-1)!}$. The *exponential* distribution has one parameter, the mean value λ^{-1}, with $\lambda \in \mathbb{R}_{>0}$. The *Bernoulli* distribution has one parameter, a probability $p \in [0, 1]$.

⟨*EnvAssumptionProb*⟩ ::= ⟨*ProbabilityExpression*⟩

⟨*ReqProb*⟩ ::= ⟨*ProbabilityBound*⟩

⟨*ProbabilityExpression*⟩ ::= ⟨*ProbabilityBound*⟩ | ⟨*ProbabilityDistribution*⟩

⟨*ProbabilityBound*⟩ ::= ⟨*ComparisonOperator*⟩ ⟨*RealNumber*⟩

⟨*ComparisonOperator*⟩ ::= ‘=’ | ‘≥’ | ‘>’ | ‘≤’ | ‘<’

⟨*RealNumber*⟩ ::= ⟨*Number*⟩ ’ . ’ ⟨*Number*⟩

⟨*Number*⟩ ::= ⟨*Digit*⟩ | ⟨*Number*⟩ ⟨*Digit*⟩

⟨*Digit*⟩ ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

⟨*ProbabilityDistribution*⟩ ::= ⟨*Uniform*⟩ | ⟨*Binomial*⟩ | ⟨*Normal*⟩ | ⟨*Poisson*⟩
| ⟨*Gamma*⟩ | ⟨*Exponential*⟩ | ⟨*Bernoulli*⟩ | ⟨*Histogram*⟩

⟨*Uniform*⟩ ::= ’uniform’ ’(’ ⟨*RealNumber*⟩ ’,’ ⟨*RealNumber*⟩ ’)’

⟨*Binomial*⟩ ::= ’binomial’ ’(’ ⟨*Number*⟩ ’)’

⟨*Normal*⟩ ::= ’normal’ ’(’ ⟨*RealNumber*⟩ ’,’ ⟨*RealNumber*⟩ ’)’

⟨*Poisson*⟩ ::= ’poisson’ ’(’ ⟨*RealNumber*⟩ ’)’

⟨*Gamma*⟩ ::= ’gamma’ ’(’ ⟨*Number*⟩ ’,’ ⟨*RealNumber*⟩ ’)’

⟨*Exponential*⟩ ::= ’exponential’ ’(’ ⟨*RealNumber*⟩ ’)’

⟨*Bernoulli*⟩ ::= ’bernoulli’ ’(’ ⟨*RealNumber*⟩ ’)’

⟨*Histogram*⟩ ::= ’histogram’ ’(’ {⟨*RealNumber*⟩ ’,’ ⟨*RealNumber*⟩}*, ⟨*RealNumber*⟩ ’)’

Grammar 4.2.: EBNF Grammar for the Probability Expression in the Extended Safety Requirements.

Table 4.6.: Mobile Service Robot: Informal and Formal Specification of the Safety Requirement extended with Environment Assumptions.

ID	Informal Specification	Formal Specification
ESR1	If dynamic obstacles execute legal lane changes in the robot's sensor horizon with the uniform probability distribution, then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	$P_{uniform(0,2)}(G\ LegalLangeChange) \rightarrow P_{\geq 0.99}(G\ \neg ActiveCollision)$
ESR2	If dynamic obstacles execute illegal lane changes in the robot's sensor horizon with a probability of at most 0.01, then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	$P_{\leq 0.01}(F\ IllegalLangeChange) \rightarrow P_{\geq 0.99}(G\ \neg ActiveCollision)$
ESR3	If dynamic obstacles follow the robot in a blind spot of the robot sensors with probability of at most 0.0001, then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	$P_{\leq 0.0001}(F\ FollowingObstacleInBlindSpot) \rightarrow P_{\geq 0.99}(G\ \neg ActiveCollision)$
ESR4	If dynamic obstacles in the robot's sensor horizon change their velocity during their drive with the histogram distribution $\{(0, 0.125), (1, 0.75), (2, 0.125), 3\}$, then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	$P_{histogram\{(0,0.125),(1,0.75),(2,0.125),3\}}(G\ ObstacleVelocityChange) \rightarrow P_{\geq 0.99}(G\ \neg ActiveCollision)$
ESR5	If temporary stationary obstacles start moving in the robot's sensor horizon with a probability of at most 0.05, then with a probability of at least 0.99 the robot shall not collide actively with any obstacle in its environment.	$P_{\leq 0.05}(F\ ObstacleMovementRestart) \rightarrow P_{\geq 0.99}(G\ \neg ActiveCollision)$

The *histogram* distribution is in fact an ordered collection of one or more pairs that identify the start of an interval and the probability that applies within that interval starting from the leftmost interval. Additionally, one end-interval value for the upper boundary of the last interval is defined (cf. [Obj05]). The EBNF grammar of the probability expression can be easily extended to account for other probability distributions. Using the grammars introduced for the path formula and for the probability expression, the extended safety requirements can be fully formally specified. Their formal specification is depicted alongside the respective informal specification in Table 4.6.

4.5. System Design

The system design of the mobile service robot in the revisited motivational example is carried out on the basis of the functional system specification introduced in Table 4.1, Table 4.2, and Table 4.3 (cf. Section 4.3), as well as the safety requirements specification introduced in Table 4.6 (cf. Section 4.4). The system design for the mobile service robot consists of the creation of the technical system model and the environment model. Section 4.5.1 and Section 4.5.2 present the design of the environment model and the technical system model respectively using the modeling language PRISM. Section 4.5.3 illustrates on an exemplary basis the design-time verification of the safety requirement extended with one of the explicit environment assumptions in Table 4.6. Section 4.5.4 analyses the environment assumption used in the design-time verification. Although this analysis is carried out on an exemplary basis, its goal is to give a method for mapping the environment assumptions at the level of the technical system model and the environment model. Using this mapping, the respective environment assumptions monitors are defined in Section 4.5.5.

4.5.1. Environment Model

The PRISM language is a state-based modeling language, which is based on the formalism of reactive modules developed by Alur and Henzinger in [AH99] (cf. Section 2.4.3). The language supports several formalisms which can be used to describe probabilistic systems, e.g., DTMC, CTMC and MDP (cf. Section 2.4.3).

In general, the environment model is represented by several modules, one module for each obstacle in the robot's environment. Since in the revisited motivational example the robot's environment is populated by the dynamic obstacle O_1 (cf. Figure 4.6 in Section 4.2), the environment model consists only of one PRISM module which models the behavior of this obstacle.

The environment model EM is designed as an MDP, which is henceforth denoted $\mathcal{M}_{EM}$. A state in $\mathcal{M}_{EM}$ is an interpretation of the state variables specific to EM. The set of variables of the environment model EM is denoted $\mathcal{V}_{EM}$. These variables describe the state of the dynamic obstacle O_1, e.g., $y_{O_1} \in \mathcal{V}_{EM}$ denotes the current lane of obstacle O_1, and are defined in the respective PRISM module.

There are initial values and value ranges defined for each of the state variables in $\mathcal{V}_{EM}$. Both the initial values of local variables as well as the interval bounds of their value ranges are defined through specific constants. Several mathematical notations are introduced in order to allow a better handling of these constants and variables throughout this analysis. The left side of the each notation carries out the name of the constant or variable as it is defined in the PRISM language which is mapped to the corresponding mathematical notation on the right side. Notice that these notations are consistent with and extend the notations introduced in Section 3.1 and Section 4.2. These notations are effective for the remainder of this work.

Listing 4.1 depicts the local variables used in the PRISM obstacle module, while Notation 4.5.1 introduces a mapping to the corresponding notations used henceforth in these thesis.

```
// ------------------------------------------------
// -- OBSTACLE MODULE
// ------------------------------------------------
module obstacle01

// Local Variables
// -----------------------------------------------------------
// position on lane and lane of obstacle 01
x01 : [0..maxpos] init x01_init;
y01 : [0..n-1] init y01_init;

// current velocity of obstacle 01
v01 : [0..v01_Max] init 0;

// state of obstacle 01 {IDLE_01, MOVING_01}
s01 : [IDLE_01..MOVING_01] init IDLE_01;

// flag is true if obstacle 01 is static and false otherwise
static01 : bool init true;

// Module Commands
// -----------------------------------------------------------
// ...

endmodule

```

Listing 4.1: Mobile Service Robot: Local Variables used to describe the State of the Environment Model.

Notation 4.5.1. The following mapping correspondence is defined for the local variables used in the PRISM obstacle module, which describe:

- the current obstacle coordinates:

$$(xO1, yO1) \overset{\text{def}}{=} (x_{O_1}, y_{O_1}),\ \ x_{O_1} \in [0, maxpos],\ \ y_{O_1} \in \Omega_{Lane}$$

- the current obstacle velocity:

$$vO1 \overset{\text{def}}{=} v_{O_1},\ \ v_{O_1} \in [0, v^{O_1}_{Max}]$$

- the current status of obstacle O_1, i.e., static or dynamic:

$$staticO1 \overset{\text{def}}{=} static_{O_1}$$

where $static_{O_1}$ is a boolean flag that evaluates to *true* if obstacle O_1 is static, and to *false* otherwise.

■

Listing 4.2 shows the constants that are used to define the initial values and the interval bounds for the local variables of the obstacle module.

```
mdp
// ------------------------------------------------
// TECHNICAL SYSTEM MODEL
// ------------------------------------------------
// ...
// ------------------------------------------------
// ENVIRONMENT MODEL
// ------------------------------------------------
// -- CONSTANTS FOR THE ENVIRONMENT GEOMETRY
// ------------------------------------------------
const int n = 3;                // number of lanes
const int maxpos = 100;         // end of the lane
// ------------------------------------------------
// -- CONSTANTS FOR THE OBSTACLE STATE
// ------------------------------------------------
const int IDLE_O1 = 0;          // obstacle O1 state
const int MOVING_O1 = 1;
// ------------------------------------------------
const int vO1_Max = 3;          // maximum specific obstacle velocity
// ------------------------------------------------
const int xO1_Init = 60;            // initial values for coordinates of obstacle O1
const int yO1_Init = 1;
// ------------------------------------------------
const double p_ILC = 0.01;  // probability for illegal lane change
const double p_RLC = 1/3;       // probability for legal right lane change
const double p_LLC = 1/3;       // probability for legal left lane change
const double p_NLC = 1/3;       // probability for keeping the lane
// ------------------------------------------------
const int mo = 1;               // rate that the obstacle is moving

const double p_MOVE = 0.05;     // probability of stationary obstacle starting to move
const double p_STOP = 0.05;     // probability of dynamic obstacle having stopped

// ------------------------------------------------
// -- OBSTACLE MODULE
// ------------------------------------------------
// ...
```

Listing 4.2: Mobile Service Robot: Constants used to describe the State of the Environment Model.

Notation 4.5.2. The following mapping correspondence is defined for the constants of the PRISM obstacle module, which describe:

- the specific maximum velocity of obstacle O_1:

$$vO1\_Max \overset{\text{def}}{=} v^{O_1}_{Max}$$

- the probability for obstacle O_1 to temporary stop or to remain at rest (STOP) and for obstacle O_1 to start or to resume its movement (MOVE):

$$p\_STOP \overset{\text{def}}{=} p_{STOP} \qquad p\_MOVE \overset{\text{def}}{=} p_{MOVE}$$

- the initial values for the obstacle's coordinates:

$$(xO1\_Init, yO1\_Init) \overset{\text{def}}{=} (x^{O_1}_{Init}, y^{O_1}_{Init}),\ \ x^{O_1}_{Init} \in [0, maxpos],\ \ y^{O_1}_{Init} \in \Omega_{Lane}$$

- the probabilities for obstacle O_1 to jump over several lanes also denoted as illegal lane change (ILC), for obstacle O_1 to execute a legal lane change to its left lane (LLC), for obstacle O_1 to carry out a legal lane change to its right lane (RLC) and for obstacle to maintain its lane also denoted as no lane change (NLC):

$$p\_ILC \overset{\text{def}}{=} p_{ILC} \qquad p\_LLC \overset{\text{def}}{=} p_{LLC}$$
$$p\_RLC \overset{\text{def}}{=} p_{RLC} \qquad p\_NLC \overset{\text{def}}{=} p_{NLC}$$

■

Dynamic obstacle O_1 is situated at its initial position $(x^{O_1}_{Init}, y^{O_1}_{Init})$ and moves towards its destination, the origin point of the Cartesian coordinate system $(0, 0)$. In order to reach its destination, obstacle O_1 employs velocity changing as well as lane changing maneuvers. These maneuvers are carried out through specific PRISM commands. A command in the PRISM modeling language has the general form shown in Equation (2.1) in Section 2.4.3.

The obstacle O_1 is modeled as a probabilistic automaton with two states: `IDLE_O1` and `MOVING_O1`. The behavior of obstacle O_1 is described by the PRISM commands in Listing 4.3, Listing 4.4, and Listing 4.5. Listing 4.3 and Listing 4.4 shows the commands for the velocity changing maneuvers for temporary stationary obstacles and respectively for dynamic obstacles in the PRISM module of obstacle O_1.

```
// ----------------------------------------------------------
// -- OBSTACLE MODULE
// ----------------------------------------------------------
module obstacleO1

// Local Variables
// ----------------------------------------------------------
// ...

// Module Commands
// ----------------------------------------------------------
// [1]. IDLE -> IDLE
// ----------------------------------------------------------
[move] (sO1=IDLE_O1) & (xO1<=0) & (staticO1)
-> mo:(sO1'=IDLE_O1);

// [2]. IDLE -> IDLE or
//      IDLE -> MOVING
// ----------------------------------------------------------
[move] (sO1=IDLE_O1) & (xO1>0) & (staticO1)
-> 0.5:(sO1'=MOVING_O1) + 0.5:(sO1'=IDLE_O1);

// [3]. MOVING -> MOVING or
//      MOVING -> IDLE
// ----------------------------------------------------------
[move] (sO1=MOVING_O1) & (xO1>0) & (staticO1) & (vO1=0) & (xO1-vO1_Max>=0)
-> 1-p_MOVE:(sO1'=IDLE_O1) & (vO1'=0) & (xO1'=xO1) & (staticO1'=true) +
1/8*p_MOVE:(sO1'=MOVING_O1) & (vO1'=1) & (xO1'=xO1-1) & (staticO1'=false) +
3/4*p_MOVE:(sO1'=MOVING_O1) & (vO1'=2) & (xO1'=xO1-2) & (staticO1'=false)+
1/8*p_MOVE:(sO1'=MOVING_O1) & (vO1'=3) & (xO1'=xO1-3) & (staticO1'=false);

// ...

endmodule
```

Listing 4.3: Mobile Service Robot: Forwards Movement Commands for Temporary Stationary Obstacles in the Environment Model.

Notice that obstacle O_1 is initially a stationary obstacle and the obstacle module is in state `IDLE_O1`. If obstacle O_1 is stationary and already at the obstacle's destination point $(0, 0)$, then the obstacle module maintains the state `IDLE_O1` (command [1] in Listing 4.3). In this state, if obstacle O_1 is not already at its destination point, it can either further remain stationary or it can become a dynamic obstacle and start to drive towards its destination point $(0, 0)$. The choice which obstacle O_1 has to make in state `IDLE_O1` is a probabilistic one, controlled by the uniform discrete probability distribution $(0.5, 0.5)$ (command [2] in Listing 4.3). During its drive, obstacle O_1 has a probabilistic choice of either becoming temporary stationary or continuing its drive towards its destination point $(0, 0)$ (command [4] in Listing 4.4). Should it become temporary stationary, obstacle O_1 is presented again with the probabilistic choice between staying stationary or becoming again a moving obstacle (command [3] in Listing 4.3). The difference between a temporary stationary obstacle and a dynamic obstacle is modeled through the boolean flag $static_{O_1}$ and the obstacle current velocity v_{O_1}, along with the state of the obstacle module. If O_1 is a dynamic obstacle, then the boolean flag $static_{O_1}$ evaluates to the truth value *true*, the current obstacle obstacle velocity is strictly a positive number, i.e. $v_{O_1} > 0$, and the obstacle module is in state `MOVING_O1`. Notice that the command [3] in Listing 4.3 and command [4] in Listing 4.4 model the requirements FR1 and FR3 from Table 4.1 in the environment model.

During its movement, obstacle O_1 chooses probabilistically from the interval $[0, v_{Max}^{O_1}]$ a value for its current velocity (command [3] Listing 4.3 and command [4] in Listing 4.4). Based on its current velocity, the new position of the obstacle O_1 is computed and updated (command [5] in Listing 4.4). The obstacle module is modeled so that, once it reaches its destination point $(0, 0)$, it comes to a full stop, i.e., the obstacle becomes definitive stationary and the obstacle module enters the state `IDLE_O1` (commands [6] and [7] in Listing 4.4).

```
        // -----------------------------------------------------------
        // -- OBSTACLE MODULE
        // -----------------------------------------------------------
        module obstacle01

        // Local Variables
        // -----------------------------------------------------------
        // ...

        // Module Commands
        // -----------------------------------------------------------
        // ...

        // [4]. MOVING -> MOVING or
        //      MOVING -> IDLE
        // -----------------------------------------------------------
        [move] (s01=MOVING_01) & (x01>0) & (!static01) & (v01>0) & (x01-v01_Max>=0)
        -> p_STOP:(s01'=IDLE_01) & (v01'=0) & (x01'=x01) & (static01'=true) +
        1/8*(1-p_STOP):(s01'=MOVING_01) & (v01'=1) & (x01'=x01-1) & (static01'=false) +
        3/4*(1-p_STOP):(s01'=MOVING_01) & (v01'=2) & (x01'=x01-2) & (static01'=false)+
        1/8*(1-p_STOP):(s01'=MOVING_01) & (v01'=3) & (x01'=x01-3) & (static01'=false);

        // [5]. MOVING -> MOVING
        // -----------------------------------------------------------
        [move] (s01=MOVING_01) & (x01>0) & (!static01) & (v01>0) & (x01-v01_Max<0) & (x01-v01>=0)
        -> mo:(s01'=MOVING_01) & (x01'=x01-v01);

        // [6]. MOVING -> MOVING
        // -----------------------------------------------------------
        [move] (s01=MOVING_01) & (x01>0) & (!static01) & (x01-v01<0)
        -> mo:(s01'=MOVING_01) & (x01'=0);

        // [7]. MOVING -> IDLE
        // -----------------------------------------------------------
        [move] (s01=MOVING_01) & (x01<=0) & (!static01)
        -> mo:(s01'=IDLE_01) & (static01'=true) & (v01'=0);

        // ...

        endmodule

```

Listing 4.4: Mobile Service Robot: Dynamic Obstacle Forwards Movement Commands in the Environment Model.

Notice that the velocity changing behavior of obstacle O_1 underlies a probability distribution defined with the constants p_{STOP} and p_{MOVE} (cf. Listing 4.2). It is also worth noting that the probability for a stationary obstacle to start moving is considered to be 0.05 as formulated in requirement FR5 of the functional system specification. Listing 4.5

shows the Prism commands with which the lane changing behavior of obstacle O_1 is modeled.

```
// ----------------------------------------------------------
// -- OBSTACLE MODULE
// ----------------------------------------------------------
module obstacle01

// Local Variables
// ----------------------------------------------------------
// ...

// Module Commands
// ----------------------------------------------------------
// ...

// [8] Probability distribution for obstacle O1 to leave lane 1
// ----------------------------------------------------------
[move] (sO1=MOVING_O1) & (vO1>0) & (yO1=1)
-> p_LLC:(yO1'=0) + p_RLC:(yO1'=2) + p_NLC:(yO1'=1);

// [9] Probability distribution for obstacle O1 to leave lane 0
// ----------------------------------------------------------
[move] (sO1=MOVING_O1) & (vO1>0) & (yO1=0)
-> p_ILC:(yO1'=2) + (1-p_ILC)/2:(yO1'=1) + (1-p_ILC)/2:(yO1'=0);

// [10] Real probability distribution for obstacle O1 to leave lane 2
// ----------------------------------------------------------
[move] (sO1=MOVING_O1) & (vO1>0) & (yO1=2)
-> p_ILC:(yO1'=0) + (1-p_ILC)/2:(yO1'=1) + (1-p_ILC)/2:(yO1'=2);

endmodule

```

Listing 4.5: Mobile Service Robot: Set of Commands for Obstacle Lane Changes in the Environment Model.

Depending on the lane on which obstacle O_1 is situated, its probabilistic lane changing behavior is governed by a different probability distribution. Initially, obstacle O_1 is situated on lane 1 (cf. Listing 4.2), which is the ego lane of the mobile service robot (cf. Listing 4.7). The lane changing behavior from lane 1 to the other two lanes is governed by a probability distribution defined through the constants p_{RLC}, p_{LLC}, and p_{NLC} (command [8] in Listing 4.5). In turn, if the obstacle O_1 is situated one of the other two lanes, lane 0 or lane 1, then the probability distribution that controls the lane changing behavior of the obstacle is defined through the constant p_{ILC} (commands [9] and [10] in Listing 4.5). All the constants pertaining to the obstacle model are defined in Listing 4.2.

$$p_{ILC} = 0.01 \qquad\qquad p_{RLC} = p_{NLC}$$

$$p_{NLC} = \begin{cases} \frac{1-(n-2)*p_{ILC}}{2}, & y_{O_1} = 0 \\ \frac{1-(n-2)*p_{ILC}}{2}, & y_{O_1} = n-1 \\ \frac{1-(n-3)*p_{ILC}}{3}, & y_{O_1} \in \{1, \dots, n-2\} \end{cases} \qquad\qquad p_{LLC} = p_{NLC}$$

Notice that the probability for an illegal lane change carried out by obstacle O_1 is considered to be 0.0001 as expressed in requirements FR7 of the functional system specification. The probability for obstacle O_1 to keep its lane as well as the probability of obstacle O_1 to carry out a legal lane change are computed on the basis of the probability of illegal lane change and are uniformly distributed.

4.5.2. Technical System Model

The technical system model is represented by a Prism module, which models the behavior of the mobile service robot. The technical system model SM is designed as an MDP, which is henceforth denoted $\mathcal{M}_{SM}$. A state in $\mathcal{M}_{SM}$ is an interpretation of the state variables specific to SM. The set of variables of the technical system model SM is denoted $\mathcal{V}_{SM}$. These variables describe the state of the mobile service robot, e.g., $y_R \in \mathcal{V}_{SM}$ denotes the current lane of the robot, and are defined in the Prism robot's module. The set of variables $\mathcal{V}_{SM}$ is divided into four distinct subsets of variables:

$$\mathcal{V}_{SM} = \mathcal{V}^{SM}_{Internal} \cup \mathcal{V}^{SM}_{Assumed} \cup \mathcal{V}^{SM}_{Observed} \cup \mathcal{V}^{SM}_{Predicted}$$

where:

- $\mathcal{V}^{SM}_{Internal} \subseteq \mathcal{V}_{SM}$ is a set of variables which describe exclusively the system's state of the technical system model, e.g., $x_R, y_R \in \mathcal{V}_{Internal}$,
- $\mathcal{V}^{SM}_{Assumed} \subseteq \mathcal{V}_{SM}$ is a set of variables that describe assumed facts about the future behavior in the system's environment, e.g., $p^{ILC}_{Assumed} \in \mathcal{V}^{SM}_{Assumed}$,
- $\mathcal{V}^{SM}_{Observed} \subseteq \mathcal{V}_{SM}$ is a set of variables which describe the system observations of the current behavior in its environment, e.g., $y^{O_1}_{Observed} \in \mathcal{V}^{SM}_{Observed}$,
- $\mathcal{V}^{SM}_{Predicted} \subseteq \mathcal{V}_{SM}$ is a set of variables which describe the system predictions of the future behavior in the system's environment, e.g., $y^{O_1}_{Predicted} \in \mathcal{V}^{SM}_{Predicted}$.

Similar to the environment model, there are initial values and value ranges defined for each of variables in $\mathcal{V}_{SM}$. The initial values of the local variables as well as the interval bounds of their value ranges are defined through specific constants. As with the environment model, several notations specific to the technical system model must be introduced in order to allow a better handling of these constants and variables throughout this analysis. In each notation on the left side there is the name of the constant or variable as it is defined in the Prism language. This name is mapped to the corresponding mathematical notation on the right side. Similar to the notations introduced for the environment model, the mathematical notations pertaining to the technical system model are consistent with and extend the notations introduced in Section 3.1 and Section 4.2. These notations are effective for the remainder of this work.

Listing 4.6 depicts the local variables used in the Prism robot module, while Notation 4.5.3 introduces a mapping to the corresponding notations used henceforth in these thesis.

```
// ----------------------------------------------------------
// -- ROBOT MODULE
// ----------------------------------------------------------
module robot

// Local Variables
// ----------------------------------------------------------
// current robot coordinates
xR : [0..maxpos] init xR_Init;  // position on lane
yR : [0..n-1] init yR_Init; // lane

// current robot velocity
vR : [0..vR_Max] init 0;

// robot state {IDLE_R, ACCEL_R, DRIVE_R, BRAKE_R, STOP_R}
sR : [IDLE_R..STOP_R] init IDLE_R;

// observed obstacle lane
yO1_Observed : [0..n-1] init yO1_Observed_Init;

// predicted obstacle lane
yO1_Predicted : [0..n-1] init yO1_Predicted_Init;

// Module Commands
// ----------------------------------------------------------
// ...

endmodule

```

Listing 4.6: Mobile Service Robot: Local Variables in the Description of the State of the Technical System Model.

Notation 4.5.3. The following mapping correspondence is defined for the local variables used in the PRISM robot module, which describe:

- the current robot coordinates:

$$(xR, yR) \overset{\text{def}}{=} (x_R, y_R),\ \ x_R \in [0, maxpos],\ \ y_R \in \Omega_{Lane}$$

- the current robot velocity:

$$vR \overset{\text{def}}{=} v_R,\ \ v_R \in [0, v^R_{Max}]$$

- the robot's observation through its sensors of the lane on which the dynamic obstacle O_1 is currently situated:

$$yO1\_Observed \overset{\text{def}}{=} y^{O_1}_{Observed},\ \ y^{O_1}_{Observed} \in \Omega_{Lane}$$

- the robot's prediction of the lane on which the dynamic obstacle O_1 will be situated in the next computation step:

$$yO1\_Predicted \overset{\text{def}}{=} y^{O_1}_{Predicted},\ \ y^{O_1}_{Predicted} \in \Omega_{Lane}$$

■

Notice that the variables which describe the robot observation $y^{O_1}_{Observed}$ and the robot prediction $y^{O_1}_{Predicted}$ of obstacle O_1's lane take values from the same sample space Ω_{Lane} as the variable y_R, which models the current lane of the robot.

Listing 4.7 shows the constants that are used to define the initial values and the interval bounds for the local variables of the robot module. Among the defined constants, there are several which are used to describe the robot's observations, the robot's predictions and its environment assumptions with respect to the lane changing behavior of the dynamic obstacle O_1.

```
mdp
// ------------------------------------------------
// TECHNICAL SYSTEM MODEL
// ------------------------------------------------
// -- CONSTANTS FOR THE ROBOT STATE
// ------------------------------------------------
const int IDLE_R = 0;                   // robot state
const int ACCEL_R = 1;
const int DRIVE_R = 2;
const int BRAKE_R = 3;
const int STOP_R = 4;
// ------------------------------------------------
const int hR = 30;                      // robot sensor horizon
// ------------------------------------------------
const int vR_Max = 5;                   // maximum specific robot velocity
// ------------------------------------------------
const int xR_Init = 0;                  // robot initial coordinates
const int yR_Init = 1;
// ------------------------------------------------
const int xR_Goal = 100;                // robot goal coordinates
const int yR_Goal = 1;
// ------------------------------------------------
const double mr = 1;                    // rate that the robot moves
// ------------------------------------------------
// -- CONSTANTS FOR THE ROBOT OBSERVATIONS
// ------------------------------------------------
const int yO1_Observed_Init = 1;        // initial value for observed obstacle lane
// ------------------------------------------------
// -- CONSTANTS FOR THE ROBOT ASSUMPTIONS
// ------------------------------------------------
const int vO_MaxAssumed = 2;            // maximum assumed obstacle velocity
const double p_ILC_Assumed = 0.01;    // assumed probability for Illegal Lane Change (ILC)
const double p_RLC_Assumed = 1/3;       // assumed probability for legal Right Lane Change (RLC)
const double p_LLC_Assumed = 1/3;       // assumed probability for legal Left Lane Change (LLC)
const double p_NLC_Assumed = 1/3;       // assumed probability for No Lane Change (NLC)
// ------------------------------------------------
// -- CONSTANTS FOR THE ROBOT PREDICTIONS
// ------------------------------------------------
const int yO1_Predicted_Init=1;         // initial value for predicted obstacle lane
// -----------------------------------------------------------
// -- ROBOT MODULE
// -----------------------------------------------------------
// ...

```

Listing 4.7: Mobile Service Robot: Constants used in the Description of the the State of the Technical System Model.

Notation 4.5.4. The following mapping correspondence is defined for the constants of the PRISM robot module, which describe:

- the robot's sensor horizon:

$$hR \overset{\text{def}}{=} h_R$$

- the specific maximum velocity of the robot:

$$vR\_Max \overset{\text{def}}{=} v^R_{Max}$$

- the initial values for the robot's coordinates:

$$(xR\_Init, yR\_Init) \overset{\text{def}}{=} (x^R_{Init}, y^R_{Init})$$

- the coordinates of the robot's destination:

$$(xR\_Goal, yR\_Goal) \overset{\text{def}}{=} (x^R_{Goal}, y^R_{Goal})$$

- the initial value for the robot's observation of obstacle O_1 lane:

$$yO1\_Observed\_Init \overset{\text{def}}{=} y^{O_1}_{ObservedInit}$$

- the maximum assumed obstacle velocity for obstacle O_1:

$$\mathit{vO\_MaxAssumed} \stackrel{\text{def}}{=} v^{O}_{\mathit{MaxAssumed}}$$

- the assumed probabilities for obstacle O_1 to execute an illegal lane change (ILC) by jumping over several lanes, for obstacle O_1 to execute a legal lane change to its left lane (LLC), for obstacle O_1 to carry out a legal lane change to its right lane (RLC), and for obstacle O_1 to maintain its lane also denoted as no lane change (NLC):

$$\begin{aligned} \mathit{p\_ILC\_Assumed} &\stackrel{\text{def}}{=} p^{\mathit{ILC}}_{\mathit{Assumed}} & \mathit{p\_LLC\_Assumed} &\stackrel{\text{def}}{=} p^{\mathit{LLC}}_{\mathit{Assumed}} \\ \mathit{p\_RLC\_Assumed} &\stackrel{\text{def}}{=} p^{\mathit{RLC}}_{\mathit{Assumed}} & \mathit{p\_NLC\_Assumed} &\stackrel{\text{def}}{=} p^{\mathit{NLC}}_{\mathit{Assumed}} \end{aligned}$$

- the initial value for the robot's prediction of obstacle O_1 lane:

$$\mathit{yO1\_Predicted\_Init} \stackrel{\text{def}}{=} y^{O_1}_{\mathit{PredictedInit}}$$

■

The behavior of the technical system model is described through the Prism commands of the robot module. An excerpt of the set of commands pertaining to the robot module is shown in Listing 4.8. The robot module has five possible states `IDLE_R`, `ACCEL_R`, `DRIVE_R`, `BRAKE_R`, and `STOP_R`. Initially, as the robot has not yet started to move, the robot module is in state `IDLE_R`. At the same time, this state is also the final state, in which the automaton enters when the robot reaches its destination. The mobile service robot is commissioned to drive towards a given destination (FR8). As long as the robot has not reached its maximum velocity, the robot module remains in the state `ACCEL_R` (FR8.1). The robot drives with maximum velocity, as long as no collision danger is detected, i.e., the robot module enters the location `DRIVE_R` and remains there for as long as possible (FR8.2). Should the robot detect a collision danger or should it approach its destination, then the robot module enters the location `BRAKE_R` (FR9.1 and FR12), which enables the robot to reduce its velocity, and eventually, come to a standstill (state `STOP_R`).

```
// -----------------------------------------------------------
// -- ROBOT MODULE
// -----------------------------------------------------------
module robot

// Local Variables
// -----------------------------------------------------------
// ...

// Module Commands
// -----------------------------------------------------------
// [1]. IDLE -> IDLE:
// -----------------------------------------------------------
[move] sR=IDLE_R & (!robot_dest_reached | collision_danger)
-> mr:(sR'=IDLE_R) & (y01_Observed'=y01);

// [2]. IDLE -> ACCELERATE:
// -----------------------------------------------------------
[move] sR=IDLE_R & (!robot_near_dest & !collision_danger & !robot_cruise_vel_reached)
-> mr:(sR'=ACCEL_R) & (y01_Observed'=y01);

// [3]. ACCELERATE -> ACCELERATE:
// -----------------------------------------------------------
[move] sR=ACCEL_R & (!robot_near_dest & !robot_cruise_vel_reached &
(!collision_danger | obstacle_passed_by)) & (xR+vR+1<maxpos)
-> mr:(sR'=ACCEL_R) & (vR'=vR+1) & (xR'=xR+vR+1) & (y01_Observed'=y01);

// ...

endmodule

```

Listing 4.8: Mobile Service Robot: Excerpt of the Set of Commands in the Technical System Model.

```
// -----------------------------------------------------------
// -- ROBOT MODULE
// -----------------------------------------------------------
module robot

// Local Variables
// -----------------------------------------------------------
// ...

// Module Commands
// -----------------------------------------------------------
// ...

// [18] Assumed probability distribution for obstacle to leave lane 0
[move] y01_Observed=0 -> p_ILC_Assumed:(y01_Predicted'=2) +
(1-p_ILC_Assumed)/2:(y01_Predicted'=1) +
(1-p_ILC_Assumed)/2:(y01_Predicted'=0);

// [19] Assumed probability distribution for obstacle to leave lane 2
[move] y01_Observed=2 -> p_ILC_Assumed:(y01_Predicted'=0) +
(1-p_ILC_Assumed)/2:(y01_Predicted'=1) +
(1-p_ILC_Assumed)/2:(y01_Predicted'=2);

// [20] Assumed probability distribution for obstacle to leave lane 1
[move] y01_Observed=1 -> p_LLC_Assumed:(y01_Predicted'=0) +
p_RLC_Assumed:(y01_Predicted'=2) +
p_NLC_Assumed:(y01_Predicted'=1);

endmodule

```

Listing 4.9: Mobile Service Robot: Assumed Probability Distribution defined in the Technical System Model for Obstacle Lane Changing Events occurring in the Environment Model.

An alternative collision avoidance maneuver to braking to a standstill in case of collision danger is changing to a safe lane. The mobile service robot is able to detect a safe lane and change to it in case of collision danger (FR12.1). This is justified by the purpose of the mobile service robot, i.e., to autonomously reach their destination and to cover as much ground as possible in the process before having to brake. Remember that a lane is considered to be safe if no visible obstacles are detected inside the region spanned by the robot's collision distance. The robot can use the opportunity of changing to a safe lane and resume accelerating in order to drive further, rather than trigger the emergency brake. However, if no lane change is possible, then the collision danger persists and the robot is forced to brake until it comes to a full stop, i.e., the robot module enters the state `STOP_R`. As long as the collision danger persists and no other target safe target lane is detected, the robot remains in a state of rest, i.e., the robot module maintains the state `STOP_R` (FR14).

Notice that even in case of collision danger and with no other safe target lane detected, the mobile service robot has a further alternative before braking to a standstill. Thus, the robot is able to check whether it can drive further, albeit with a reduced velocity (FR13.1 and FR13.2). This accounts for the possibility that the obstacle representing the collision danger may reduce its speed, thus allowing the robot to drive further with reduced velocity, rather than braking to a full stop. If it cannot drive further with reduced velocity, then the robot applies the brakes until it reaches a full stop (FR13.3). While the robot is braking or after the robot has reached a standstill due to collision danger, it may happen that another lane becomes safe to move on (FR8.4 and FR8.5). The robot then changes to the safe target lane and continues its drive until it reaches its destination.

In order to assess which lane is safe, the robot module uses predictions of the lane on which obstacle O_1 will find itself in the next computation step based on the robot's observations of the obstacle lane in the current step. Notice that the robot updates its observations of the obstacle lane at each transition that takes place between the between the states `IDLE_R`, `ACCEL_R`, `DRIVE_R`, `BRAKE_R`, and `STOP_R` of the robot module. Listing 4.9 shows the Prism commands of the robot module which are used to compute the robot's predictions with respect to the obstacle lane. Notice that the computation of the robot's prediction of the obstacle lane occurs probabilistically based on assumed probability distributions, that define the probability of an illegal lane change carried out by obstacle O_1, the probabilities that the obstacle O_1 executes a legal lane change to the right lane and respectively to the left lane, as well as the probability that the obstacle O_1 keeps its current lane.

4.5.3. Design Time Verification

For the design-time verification, the overall system model is verified against the extended safety requirement ESR2, which is formally specified in Table 4.6. The overall system model, consisting of the parallel composition between the technical system model and the environment model, is modeled as an MDP in the Prism modeling language. The extended safety requirement ESR2, formally specified in Table 4.6, is translated as a multi-objective property in PCTL and verified with the Prism model checker. Table 4.7 shows the multi-objective property alongside the formal specification of ESR2, together with its verification result.

Table 4.7.: Mobile Service Robot: Verification of Extended Safety Requirement ESR2 expressed as Multi-objective Property.

ID	Formal Specification	Multi-objective Property	Verification Result
ESR2	$P_{\leq 0.01}(F\ \mathit{IllegalLangeChange}) \rightarrow P_{\geq 0.99}(G\ \neg \mathit{ActiveCollision})$	multi(P<= 0.01 [F *IllegalLaneChange*], P>=0.99 [G ¬ *ActiveCollision*)	✓

4.5.4. Analysis of the Environment Assumptions

This section carries out an analysis of the environment assumptions defined explicitly during the system design-time. The goal of this analysis is to map the environment assumptions on the technical system model SM and the environment model EM and use this mapping to construct the respective environment assumptions monitors.

This analysis is carried out on an exemplary basis, i.e., not all environment assumptions are taken under analysis. The considered example is environment assumption of the extended safety requirement ESR2 in Table 4.6, which covers the illegal lane changing behavior of obstacle O_1. For the sake of simplicity, ESR2 is reiterated in Equation (4.35):

$$P_{\leq 0.01}(F\ \mathit{IllegalLangeChange}) \rightarrow P_{\geq 0.99}(G\ \neg \mathit{ActiveCollision}) \tag{4.35}$$

Remember that informally, ESR2 requires that if the visible dynamic obstacle O_1 executes an illegal lane change with a probability of at most 0.01, then with a probability of at least 0.99 the robot shall not cause an active collision with obstacle O_1.

Using Definition 4.4.3 and Definition 4.4.1 of the first-order predicates for illegal lane change and active collision, Equation (4.35) can be rewritten as Equation (4.36):

$$\begin{aligned} P_{\leq 0.01}[F\ (|y_{O_1}(t) - y_{O_1}(t+1)| > 1)] \rightarrow & P_{\geq 0.99}[G\ ((v_R == 0) \vee \neg((y_R == y_{O_1}) \wedge \\ & (-c_R \leq x_{O_1} - x_R) \wedge (x_{O_1} - x_R \leq c_R)))] \end{aligned} \tag{4.36}$$

where $y_{O_1}(t)$ and $y_{O_1}(t+1)$ represent the current lane of obstacle O_1 at the consecutive steps t and respectively $t+1$.

In order identify the elements of ESR2 and understand their semantic, the rewritten ESR2 in Equation (4.36) must be matched with the general formal specification for extended safety requirements given in Definition 4.4.7 in Section 4.4.5. Thus, the following elements of ESR2 are identified:

- $F\ (|y_{O_1}(t) - y_{O_1}(t+1)| > 1)$ represents the environment assumption clause,
- 0.01 is the upper bound of the environment assumption's probability,
- $G\ ((v_R == 0) \vee \neg((y_R == y_{O_1}) \wedge (-c_R \leq x_{O_1} - x_R) \wedge (x_{O_1} - x_R \leq c_R)))$ represents the main clause of the extended safety requirement ESR2, and
- 0.99 is the lower bound for the probability of main clause of ESR2.

Notice that the environment assumption $P_{\leq 0.01}[F\left(|y_{O_1}(t) - y_{O_1}(t+1)| > 1\right)]$ can be rewritten as:

$$P\left[F\left(|y_{O_1}(t) - y_{O_1}(t+1)| > 1\right)\right] \leq 0.01 \tag{4.37}$$

In Equation (4.37), the upper bound for the environment assumption is computed through the probability distribution function P^{SM}_{Lane} that is defined in the technical system model SM. This function is used by the robot to compute the assumed probabilities of occurrence for the events which describe the lane changing behavior of obstacle O_1 in its environment. The probability distribution function P^{SM}_{Lane} is defined in Equation (4.38):

$$\begin{aligned} &P^{SM}_{Lane} : Dom(y^{O_1}_{Observed}) \times Dom(y^{O_1}_{Predicted}) \rightarrow [0,1] \\ &P^{SM}_{Lane}(y^{O_1}_{Observed}(t), y^{O_1}_{Predicted}(t+1)) = \begin{cases} p^{NLC}_{Assumed}, & \Delta^{SM}_{Lane} = 0 \\ p^{LLC}_{Assumed}, & \Delta^{SM}_{Lane} = 1 \\ p^{RLC}_{Assumed}, & \Delta^{SM}_{Lane} = -1 \\ p^{ILC}_{Assumed}, & |\Delta^{SM}_{Lane}| > 1 \end{cases} \end{aligned} \tag{4.38}$$

where $\Delta^{SM}_{Lane} = y^{O_1}_{Observed}(t) - y^{O_1}_{Predicted}(t+1)$. Notice that $Dom(y^{O_1}_{Observed})$ and $Dom(y^{O_1}_{Predicted})$ are the domains of the variables $y^{O_1}_{Observed}$ and respectively $y^{O_1}_{Predicted}$, with $Dom(y^{O_1}_{Observed}) = \Omega_{Lane}$ and $Dom(y^{O_1}_{Predicted}) = \Omega_{Lane}$.

The probability distribution function P^{SM}_{Lane} is modeled with state variables of the technical system model SM. Thus, P^{SM}_{Lane} takes as input the lane of obstacle O_1 at step t as observed by the robot, $y^{O_1}_{Observed} \in \mathcal{V}^{SM}_{Observed}$, and the lane of obstacle O_1 at step $t+1$ as predicted by the robot, $y^{O_1}_{Predicted} \in \mathcal{V}^{SM}_{Predicted}$. The output of the probability distribution function is modeled with the constants $p^{ILC}_{Assumed}, p^{LLC}_{Assumed}, p^{RLC}_{Assumed}, p^{NLC}_{Assumed} \in \mathcal{V}^{SM}_{Assumed}$, which represent the assumed probabilities for obstacle O_1 to execute an illegal lane change, to execute a legal lane change to its left lane, to carry out a legal lane change to its right lane, and to maintain its lane also denoted as no lane change.

From a functional point of view, the semantic of the probability distribution function P^{SM}_{Lane} is as follows. The robot observes that obstacle O_1 is situated on lane $y^{O_1}_{Observed}$ at step t and predicts that the obstacle will occupy the lane $y^{O_1}_{Predicted}$ at computation step $t+1$. Notice that, in case $\Delta^{SM}_{Lane} \neq 0$, the probability function P^{SM}_{Lane} denotes the assumed probability with which obstacle O_1 changes from the observed lane $y^{O_1}_{Observed}$ to the predicted lane $y^{O_1}_{Predicted}$ in the next computation step $t+1$. In case $\Delta^{SM}_{Lane} = 0$, then the probability function P^{SM}_{Lane} expresses the assumed probability with which obstacle O_1 maintains its current lane in the next computation step $t+1$.

The environment assumption clause in Equation (4.37) asks whether obstacle O_1 eventually executes an illegal lane change. The lane changing behavior of obstacle O_1 is governed by the probability distribution function P^{EM}_{Lane}, which is used in the environment model EM. The probability distribution function is defined in Equation (4.39):

$$P^{EM}_{Lane} : Dom(y_{O_1}) \times Dom(y_{O_1}) \rightarrow [0,1]$$
$$P^{EM}_{Lane}(y_{O_1}(t), y_{O_1}(t+1)) = \begin{cases} p_{NLC}, & \Delta^{EM}_{Lane} = 0 \\ p_{LLC}, & \Delta^{EM}_{Lane} = 1 \\ p_{RLC}, & \Delta^{EM}_{Lane} = -1 \\ p_{ILC}, & |\Delta^{EM}_{Lane}| > 1 \end{cases} \tag{4.39}$$

where $\Delta^{EM}_{Lane} = y_{O_1}(t) - y_{O_1}(t+1)$. Notice that $Dom(y_{O_1})$ is the domain of the variable y_{O_1} with $Dom(y_{O_1}) = \Omega_{Lane}$.

The probability distribution function P^{EM}_{Lane} is modeled with state variables of the environment model EM. Thus, P^{EM}_{Lane} takes as input the lane of obstacle O_1 at steps t and $t+1$, $y_{O_1} \in \mathcal{V}_{EM}$. The output of the probability distribution function is modeled with the constants $p_{ILC}, p_{LLC}, p_{RLC}, p_{NLC} \in \mathcal{V}_{EM}$, which represent probabilities of occurrence for obstacle O_1 to execute an illegal lane change, a legal lane change to its left lane, a legal lane change to its right lane, or to maintain its current lane also denoted as no lane change.

From a functional point of view, the semantic of the probability distribution function P^{EM}_{Lane} is as follows. Obstacle O_1 is situated on its current lane y_{O_1} at step t. Notice that, in case $\Delta^{EM}_{Lane} \neq 0$, the probability function P^{EM}_{Lane} denotes the probability modeled in the environment model EM with which obstacle O_1 changes from its current lane to another lane in the next computation step $t+1$. In case $\Delta^{EM}_{Lane} = 0$, then the probability function P^{EM}_{Lane} expresses the modeled probability in the environment model EM with which obstacle O_1 maintains its current lane in the next computation step $t+1$.

In this way the environment assumption in Equation (4.37) can be rewritten as:

$$P^{EM}_{Lane}(y_{O_1}(t), y_{O_1}(t+1)) \leq P^{SM}_{Lane}(y^{O_1}_{Observed}(t), y^{O_1}_{Predicted}(t+1)) \tag{4.40}$$

The environment assumption in Equation (4.37) is in fact an instance of the more general form in Equation (4.41). In Equation (4.41), X represents a feature of the environment which can be observed and about which assumptions can be made during system design-time, e.g., the lane of an obstacle in the environment of the mobile service robot. For the environment feature X, the current value, the observed value and the predicted value are modeled by the variables x, $x_{Observed}$, and respectively $x_{Predicted}$.

$$P^{EM}_{X}(x(t), x(t+1)) \leq P^{SM}_{X}(x_{Observed}(t), x_{Predicted}(t+1)) \tag{4.41}$$

4.5.5. Formal Definition of Environment Assumptions Monitors

Environment assumptions monitors are used to check the validity of environment assumptions during system test and requirements validation. The environment assumptions monitors M_{EA} run in parallel to the implemented system IS and make observations of the controlled environment CE. During the observation of the controlled environment, an environment assumptions monitor produces a specific monitor observation trace, which is then checked against the respective environment assumption.

In order to model the monitor observations, the environment assumptions monitor can introduce its own set of variables. To give a general definition for the environment assumptions monitors, further notations are introduced that are considered effective hereafter in this thesis.

Notation 4.5.5. The set of variables of the implemented system IS is denoted $\mathcal{V}_{IS}$. Similarly to the technical system model SM, the $\mathcal{V}_{IS}$ is partitioned into the following subsets:

$$\mathcal{V}_{IS} = \mathcal{V}^{IS}_{Internal} \cup \mathcal{V}^{IS}_{Assumed} \cup \mathcal{V}^{IS}_{Observed} \cup \mathcal{V}^{IS}_{Predicted}$$

where:

- $\mathcal{V}^{IS}_{Internal} \subseteq \mathcal{V}_{IS}$ is a set of variables which describe exclusively the state of the implemented system IS,
- $\mathcal{V}^{IS}_{Assumed} \subseteq \mathcal{V}_{IS}$ is a set of variables which describe assumed facts about the future behavior in the environment of the implemented system IS.
- $\mathcal{V}^{IS}_{Observed} \subseteq \mathcal{V}_{IS}$ is a set of variables which describe the observations of the implemented system IS of the current behavior in its environment.
- $\mathcal{V}^{IS}_{Predicted} \subseteq \mathcal{V}_{IS}$ is a set of variables which describe the predictions of the implemented system IS of the future behavior of the system's environment.

Since the implemented system IS is a concrete implementation of the technical system model SM (cf. Figure 4.2 in Section 4.1), the following relation exist between their respective set of variables:

$$\begin{aligned}
\mathcal{V}^{SM}_{Internal} &\subseteq \mathcal{V}^{IS}_{Internal}\\
\mathcal{V}^{SM}_{Assumed} &\subseteq \mathcal{V}^{IS}_{Assumed}\\
\mathcal{V}^{SM}_{Observed} &\subseteq \mathcal{V}^{IS}_{Observed}\\
\mathcal{V}^{SM}_{Predicted} &\subseteq \mathcal{V}^{IS}_{Predicted}
\end{aligned}$$

The set of variables which model the relevant features of the controlled environment CE is denoted $\mathcal{V}_{CE}$. Since the controlled environment CE is a concrete realization of the environment model EM (cf. Figure 4.2 in Section 4.1), the following relation exist between their respective set of variables:

$$\mathcal{V}_{EM} \subseteq \mathcal{V}_{CE}$$

The set of variables introduced by the environment assumption monitor M_ψ is denoted $\mathcal{V}_{M_\psi}$. ■

Based on the analysis carried out in the previous section, this section proposes a method to derive runtime monitors for environment assumptions. The method is shown in Figure 4.12 and is rooted in the RMEA concept depicted in Figure 4.2 (cf. Section 4.1.2).

$$\begin{tikzcd}
\mathcal{V}^{SM}_{Assumed} \arrow[d, dash, "\subseteq"'] & \ni & x^{SM}_{Assumed} \arrow[rr, dash, "\psi"] \arrow[d, dash, "="'] & & x_{EM} & \in & \mathcal{V}_{EM} \arrow[d, dash, "\subseteq"] \\
\mathcal{V}^{IS}_{Assumed} & \ni & x^{IS}_{Assumed} \arrow[r, dash, "M_{\psi}"] & x^{M_{\psi}}_{Observed} \arrow[r, dash, "obs"] & x_{CE} & \in & \mathcal{V}_{CE}
\end{tikzcd}$$

Legend:

SM – system model IS – implemented system $x^{SM}_{Assumed} \in \mathcal{V}^{SM}_{Assumed}$ - variable defined in SM that describes an assumed fact about the future behavior of feature X in the environment model $x^{IS}_{Assumed} \in \mathcal{V}^{IS}_{Assumed}$ - variable defined in IS that describes an assumed fact about the future behavior of feature X in the controlled environment	EA – set of environment assumptions $\psi \in EA$ – an environment assumption formulated about the environment feature X M_{EA} – set of runtime monitors for the environment assumptions EA $M_{\psi} \in M_{EA}$ – runtime monitor for the environment assumption ψ $x^{M_{\psi}}_{Observed} \in \mathcal{V}_{M_{\psi}}$- variable defined additionally in the environment assumptions monitor M_{ψ} to model the monitor's observation of variable x_{CE}	EM – environment model CE – controlled environment $x_{EM} \in \mathcal{V}_{EM}$ - variable defined in EM that describes the behavior of feature X in the environment model $x_{CE} \in \mathcal{V}_{CE}$ - variable defined during system test that describes the behavior of feature X in the controlled environment obs – a function that maps the variable x_{CE} onto the variable $x^{M_{\psi}}_{Observed}$

Figure 4.12.: Construction Method for Runtime Monitors of Environment Assumptions.

At design-time, the environment assumption ψ is explicitly formulated using variables defined in the technical system model and in the environment model. Specifically, for an environment feature X the environment assumption ψ uses the variable $x^{SM}_{Assumed} \in \mathcal{V}^{SM}_{Assumed}$ and the variable $x_{EM} \in \mathcal{V}_{EM}$ to compare the future behavior assumed about feature X in the technical system model with the actual behavior of feature X described in the environment model.

In system test, the runtime monitor M_{ψ} of the environment assumption ψ introduces additional variables in order to observe the behavior of environment feature X in the controlled environment. The behavior of feature X in the controlled environment is encoded through the variable x_{CE}, and its observation is carried out in the runtime monitor M_{ψ} through the variable $x^{M_{\psi}}_{Observed} \in \mathcal{V}_{M_{\psi}}$. Thus, the runtime monitor M_{ψ} compares the future behavior assumed about environment feature X in the implemented system with the actual behavior of feature X exhibited in the controlled environment. Notice that since the implemented system is a concrete realization of the technical system model, the environment assumptions defined explicitly and used during system design are also adopted in the system implementation, i.e., $x^{IS}_{Assumed} = x^{SM}_{Assumed}$. Based on the construction method depicted in Figure 4.12, Definition 4.5.1 gives a general description of runtime monitors for environment assumptions.

Definition 4.5.1 - Environment Assumptions Monitor
Let X be a feature of the controlled environment CE. Let $x_{EM} \in \mathcal{V}_{EM}$ be a variable which describes the behavior of feature X in the environment model EM. Let $x^{SM}_{Assumed} \in \mathcal{V}^{SM}_{Assumed}$ be a variable which describes in the technical system model SM an assumed fact about the behavior of X in the environment model EM. Let $x^{IS}_{Assumed} \in \mathcal{V}^{IS}_{Assumed}$ be

a variable which describes in the implemented system IS an assumed fact about the behavior of environment feature X in CE. Let $x_{CE} \in \mathcal{V}_{CE}$ be a variable which describes the behavior of feature X in CE. Let $\bowtie$ be a binary comparison operator in $\{\leq, <, \geq, >\}$. Let ψ be an environment assumption defined as:

$$\psi : x^{SM}_{Assumed} \bowtie x_{EM} \tag{4.42}$$

Then, the corresponding *environment assumption monitor* M_ψ is defined as:

$$M_\psi : x^{IS}_{Assumed} \bowtie x^{M_\psi}_{Observed} \tag{4.43}$$

where $x^{M_\psi}_{Observed} \in \mathcal{V}_{M_\psi}$ represents the observation of x_{CE} through the function obs:

$$\begin{aligned} &obs : Dom(x_{CE}) \rightarrow Dom(x^{M_\psi}_{Observed}) \\ &x^{M_\psi}_{Observed} = obs(x_{CE}) \end{aligned} \tag{4.44}$$

■

4.6. System Implementation

This section presents the implementation of the artifacts developed in system design: implementation of the technical system model (Section 4.6.1), realization of the environment model (Section 4.6.2), and implementation of the environment assumption monitors (Section 4.6.3).

4.6.1. Implementation of the System Model

The technical system model is described in Section 4.5.2 and, together with the environment model is part of the overall system model. The technical system model describes the behavior of the mobile service robot. Remember that the robot uses an assumed probability distribution to make predictions about the lane changing behavior of the dynamic obstacle O_1 in its environment. Based on the assumed probability distribution for different lane changing events and the current observed state of obstacle O_1, the robot predicts the lane on which the dynamic obstacle might move in the next computation step (cf. Section 4.5.2). The robot predictions are used in the calculation of the robot's collision distance and its determination if a collision danger exists, which in turn informs the robot's subsequent actions, e.g., braking in case of collision danger.

The implementation of the technical system model is realized with the help of the ROS 2 framework (cf. [MFG+22]). The behavior described by the MDP of the mobile service robot is implemented in a C++ ROS 2 node. This ROS 2 node controls a Robotino mobile platform simulated in the Gazebo[3] simulation software.

[3] http://gazebosim.org/

4.6.2. Realization of the Environment Model

The environment model is described in Section 4.6.2 and is part of the overall system model. The environment model describes the behavior of the obstacles in the environment of the mobile service robot. The motivational example presented in Section 4.2 features a single dynamic obstacle in the robot's environment. Remember that the dynamic obstacle chooses in a nondeterministic manner between forwards movement on its current lane, changing to another lane, and stopping. These events are in turn governed by specific probability distributions. Therefore, the formalism used for the description of the environment model is an MDP. A C++ ROS 2 node implements the behavior of the dynamic obstacle described by this MDP.

The physical environment, consisting of a road with three lanes is built in the simulation software GAZEBO. Similar to the mobile service robot, the dynamic obstacle is represented by a Robotino[4] mobile platform placed in the physical environment simulated in GAZEBO. The Robotino platform representing the dynamic obstacle is controlled by the ROS 2 node that implements the obstacle's behavior.

4.6.3. Realization of the Environment Assumptions Monitors

In the revisited motivational example, the environment assumptions defined during system design are probabilistic in order to account for the behavioral uncertainty in the system's environment. The method presented in Section 4.5.5 for the construction of environment assumptions monitors is applicable to non-probabilistic and probabilistic environment assumptions. Consider as an example the lane changing behavior of obstacle O_1, and specifically the environment assumption with respect to illegal lane change events produced by obstacle O_1. The environment assumptions monitor observing illegal lane change events compares the assumed probability of occurrence with the relative frequency of occurrence of such events in the environment of the mobile service robot. Thus, for the realization of runtime monitors for probabilistic environment assumptions, the notion of observed relative frequency of occurrence must be introduced. This notion is described in Definition 4.6.1, while in Definition 4.6.2 it is shown how the notion of observed relative frequency of occurrence is used in the realization of runtime monitors for probabilistic environment assumptions.

Definition 4.6.1 - Observed Relative Frequency of Occurrence
Let X be a feature in the controlled environment CE. Let $x_{CE} \in \mathcal{V}_{CE}$ be a variable which describes the behavior of feature X in CE. Let $e^i_{x_{CE}}$ be a specific event with respect to x_{CE}. Let $n_{Observed}(e^i_{x_{CE}})$ be the number of times that event $e^i_{x_{CE}}$ has been observed in the controlled environment CE throughout the observations history. Then the *observed relative frequency of occurrence* for the event $e^i_{x_{CE}}$ in the environment CE is computed as

[4] https://robots.ros.org/robotino/

the ratio between the number of event observations of $e^i_{x_{CE}}$ and the number of all event observations regarding x_{CE} in the observations history:

$$f_{Observed} = \frac{n_{Observed}(e^i_{x_{CE}})}{\sum_j n_{Observed}(e^j_{x_{CE}})}, i, j \in \mathbb{N} \tag{4.45}$$

■

Definition 4.6.2 - Realization of Runtime Monitors for Probabilistic Environment Assumptions

Let X be a feature of the controlled environment CE. Let $x_{EM} \in \mathcal{V}_{EM}$ be a variable which describes the behavior of feature X in the environment model EM. Let $e^i_{x_{EM}}$ be a specific event modeled in the environment model EM with respect to x_{EM}. Let $p^{SM}_{Assumed} \in \mathcal{V}^{SM}_{Assumed}$ be the probability of occurrence for the event $e^i_{x_{EM}}$ assumed in the technical system model SM. Let p_{EM} be the probability of occurrence for the event $e^i_{x_{EM}}$ modeled in the environment model EM. Let $\bowtie$ be a binary comparison operator in $\{\leq, <, \geq, >\}$. Let ψ be a probabilistic environment assumption formulated with respect to the event $e^i_{x_{EM}}$:

$$\psi : p^{SM}_{Assumed} \bowtie p_{EM} \tag{4.46}$$

Let $x_{CE} \in \mathcal{V}_{CE}$ be a variable which describes the behavior of feature X in CE. Let $e^i_{x_{CE}}$ be a specific event in the controlled environment CE corresponding to the event $e^i_{x_{EM}}$ modeled in the environment model EM. Let $p^{IS}_{Assumed} \in \mathcal{V}^{IS}_{Assumed}$ be the probability of occurrence for the event $e^i_{x_{CE}}$ assumed in the implemented system IS. Let $f^{M_\psi}_{Observed} \in \mathcal{V}_{M_\psi}$ be the observed relative frequency of occurrence for the event $e^i_{x_{CE}}$. Then the runtime monitor M_ψ of the environment assumption ψ is realized by comparing the assumed probability of occurrence with the observed relative frequency of occurrence for the event $e^i_{x_{CE}}$:

$$M_\psi : p^{IS}_{Assumed} \bowtie f^{M_\psi}_{Observed} \tag{4.47}$$

■

The monitor is built as a separate ROS2 node which runs in parallel to the ROS2 node that controls the mobile service robot. The monitor makes observations of the robot's environment by taking in the sensor data received by the robot and evaluates if the respective environment assumption is still valid or not.

4.7. System Test

In Section 4.1.2, two quality assurance goals have been specified for the phase of system test: test of the implemented system, which is discussed in Section 4.7.1, and test of the environment assumptions monitors M_{EA}, which is presented in Section 4.7.2.

4.7.1. Testing the Implemented System

During system test, the implemented system IS is stimulated with test input data and executed in the controlled environment CE. The goal is to check whether IS satisfies the safety requirement ϕ throughout its execution in CE. The test input data is provided by test cases designed by test engineers during the system test phase. Another approach to built relevant test cases is through counterexample generation (cf. Section 2.5.2).

For the revisited motivational example the test cases are generated via model-checking using the safety requirement specification $P_{\geq 0.99}(G\ \neg \mathit{ActiveCollision})$. Table XXX depicts the test cases used for the test of the implemented system.

4.7.2. Testing Environment Assumptions Monitors

The environment assumptions monitors M_{EA} serve to check if the set of environment assumptions EA defined at design-time are valid during the execution of the implemented system IS in the controlled environment CE. During system design, the explicitly defined environment assumptions have been used in the design-time verification of the technical system model SM with respect to the safety requirement ϕ (cf. Section 4.1.2). Testing the environment assumption monitors M_{EA} involves the construction of test cases that reflect behaviors of the environment which would invalidate the respective environment assumptions.

For the environment assumption $P_{\leq 0.01}(F\,\mathit{IllegalLangeChange})$ the runtime monitor is defined as follows:

The test cases defined for the environment assumptions monitor are depicted in Table YYY:

4.8. Requirements Validation

Two quality assurance goals have been defined for the requirements validation phase in Section 4.1.2: validation of the safety requirement ϕ and validation of the set of environment assumptions EA. Rather than performing validation of both the safety requirement ϕ and the environment assumptions EA, the concept of this thesis uses runtime monitoring to carry out only the validation of the environment assumptions EA (cf. Section 4.8.1). The rationale behind this is as follows: in case the environment assumptions are valid, then due to the design-time verification results and the system test results, the system fulfills also the system safety requirements. If the environment assumptions are not valid then no statement can be made as to whether the system fulfills its safety requirement or not.

4.8.1. Environment Assumptions Validation via Runtime Monitoring

For the validation of the environment assumptions, the system implementation described in Section 4.6.2 is considered to be the tested system TS. In turn, a single dynamic obstacle is present in the operational environment OE, whose lane changing behavior is not governed by a given probability distribution as in the controlled environment *CE*. Instead, the dynamic obstacle randomly changes between the three lanes of the environment simulated in Gazebo. The validation of the environment assumptions defined at system's design-time is done exemplary for the environment assumption $P_{\leq 0.01}(F\,IllegalLangeChange)$. The corresponding monitor requires that the relative frequency of occurrence with which the dynamic obstacle eventually carries out an illegal lane change in the robot's environment is at most 0.01. The environment assumption monitor is implemented as a C++ ROS 2 node, which runs in parallel to the ROS2 node that controls the mobile service robot. The monitor accesses the input sensor data received by the robot from its environment and computes continuously the relative frequency that the dynamic obstacle executes an illegal lane change. The relative frequency is computed over a maximum number of 200 seconds.

4.9. Summary

The goals of this chapter were (1) to find a method to explicitly and formally specify environment assumptions at design-time and (2) to develop an method to build runtime monitors using the formal specification of the environment assumptions defined at design-time.

The **method for the explicit and formal specification of environment assumptions** is integrated with the system development process (cf. Section 4.1.1) and consists of the following steps that were presented in detail in Section 4.4:

1. Definition of functional system requirements through the requirements elicitation and analysis phase carried out on a high-level description of the system under development,
2. Identification of safety hazards through a HARA analysis carried out on the high-level description of the system under development,
3. Identification of the safety requirements and the environment assumptions that cover the respective safety hazards,
4. Extension of the system safety requirements with the environment assumptions using a specifically designed requirements pattern,
5. Formal specification of environment assumptions using a fragment of PCTL,
6. Design of the technical system model and the environment model using functional system requirements and extended safety requirements as input, and
7. Design-time verification of the technical system model and the environment model against the system safety requirements under the consideration of the explicitly defined environment assumptions.

The **method for the construction of runtime monitors for environment assumptions** is rooted in the RMEA concept (cf. Section 4.1.2) and consists of the following steps that were presented in detail in Section 4.5:

1. Analysis of the environment assumptions on an exemplary basis,
2. Mapping the environment assumptions on the technical system model and the environment model, i.e., identifying the variables defined in the technical system

model and respectively the environment model that contribute to the formulation of the respective environment assumptions,

3. Mapping the variables identified in the technical system model on the implemented system,
4. Definition of variables that model the observations of the controlled environment carried out by the runtime monitor, and
5. Formal definition of the environment assumption monitor using first-order logic.

Chapter 5.

Case Studies

5.1. Case Study 1: Mobile Service Robot 198
5.1.1. Evaluation in the Operational Environment in the iserveU Project 198
5.2. Case Study 2: Automotive System Function for Accurate Vehicle Speed Estimation 203
5.2.1. Overview of Vehicle Speed Estimation Function 204
5.2.2. Requirements Elicitation and Analysis 205
5.2.3. Safety Analysis 206
5.2.4. System Design 213
5.2.5. System Implementation 225
5.2.6. System Test 225
5.2.7. Requirements Validation 229
5.3. Summary 231

In Section 3.10.2 the research questions which are to be tackled by this work were introduced and motivated. Chapter 4 addressed RQ-1 and RQ-2 and proposed the RMEA concept as an answer to the two research questions, which consists of a method for the explicit and formal definition of environment assumptions and a method for using this formal definition for the construction of environment assumptions monitors.

The goal of this chapter is to address the research RQ-3 and evaluate the feasibility of the RMEA concept on real-world systems. In order to achieve this goal, this chapter produces the following two artifacts as output:

Evaluation of the RMEA Concept on a Mobile Service Robot. The first artifact is the evaluation of the RMEA concept on the first case study described in this chapter: a mobile service robot method commissioned to to carry out a transportation task autonomously towards a predefined destination. This artifact contributes directly to research question RQ-3.

Evaluation of the RMEA Concept on an Automotive System Function. The second artifact is the evaluation of the RMEA concept on the second case study presented in this chapter: an automotive system function in charge of estimating and displaying the speed of a moving vehicle on its instrument board. This artifact contributes directly to research question RQ-3.

In order to create these artifacts, the chapter is structured as follows. Section 5.1 presents the first case study built around the mobile service robot, while Section 5.2 describes the second case study constructed around the vehicle speed estimation function. Section 5.3 concludes the chapter with a summary of the evaluation results in the two presented case studies.

5.1. Case Study 1: Mobile Service Robot

The first case study of this thesis is carried out on the basis of a mobile service robot commissioned to drive autonomously to a given destination without actively causing any collision with obstacles in its physical environment. Since the modeling of the robot's behavior and that of its environment as well as the implementation of the environment assumptions monitors has been discussed in the previous chapters, this section focuses on the evaluation of the environment assumptions monitors in the operational environment (Section 5.1.1).

5.1.1. Evaluation in the Operational Environment in the iserveU Project

The evaluation took place in the scope of a use case defined in the iserveU project[1]. Before going deeper in the evaluation of the environment assumption monitor, a short presentation of the iserveU project, its goal and use cases, is in order. This project had as a goal the prototype development of a mobile service robot as an intelligent transport system and the necessary technologies for the execution of logistics tasks in a hospital environment (cf. [GRR+16]). Four use cases were considered in the scope of the project (cf. [GRR+16]):

1. planned repetitive transport tasks,
2. unplanned (ad-hoc) transport tasks,
3. guidance services and cooperative transport tasks, and
4. remote management and control to support the mobile service robot in case unforeseen situations appear in the environment that are not manageable by the robot.

In the first use case, planned repetitive tasks were transportation tasks that were planned in advance and executed on a regular basis and for which the pick-up and the drop-off points as well as the scope of the transport were fixed, e.g., transport of goods deliveries from the truck loading bay to the hospital's warehouse (cf. [GRR+16]). The second use case addressed unplanned (ad-hoc) transport tasks. In contrast to planned repetitive tasks, such tasks could not be planned in advance and were related to the transportation of consumable supplies, e.g., fetching medical material or a crate of water from the hospital's warehouse to a hospital ward (cf. [GRR+16]). The third use case

[1] The full name of the iserveU project is "Intelligente modulare Serviceroboter-Funktionalitäten im menschlichen Umfeld am Beispiel von Krankenhäusern" (*Engl.:* "Intelligent modular functionalities for service robots in a human environment using hospitals as an example")

dealt with guidance services and cooperative transport tasks which the mobile service robot was required to carry out inside the hospital. Cooperative transport tasks referred to tasks which the mobile service robot executed in cooperation with hospital personal, e.g., helping nurses deliver lunch to the patients on the hospital ward. Guidance services represented services offered through the mobile service robot by which the robot escorted patients to and from diagnostic rooms and therapy stations (cf. [GRR$^+$16]).

The forth use case defined in the scope of the iserveU project focused on remote management and control, which was applied to support the mobile service robot in any unforeseen situation that appeared in the robot's environment and which the robot was not able to manage on its own. The scope of the forth use case has been used to define an environment assumptions monitor for the mobile service robot and evaluate it in the robot's operational environment. In the iserveU project, the operational environment of the mobile service robot has been restricted to one aisle of a hospital ward in Katharinenhospital in Stuttgart (cf. Figure 5.1), which was populated both with static and dynamic obstacles.

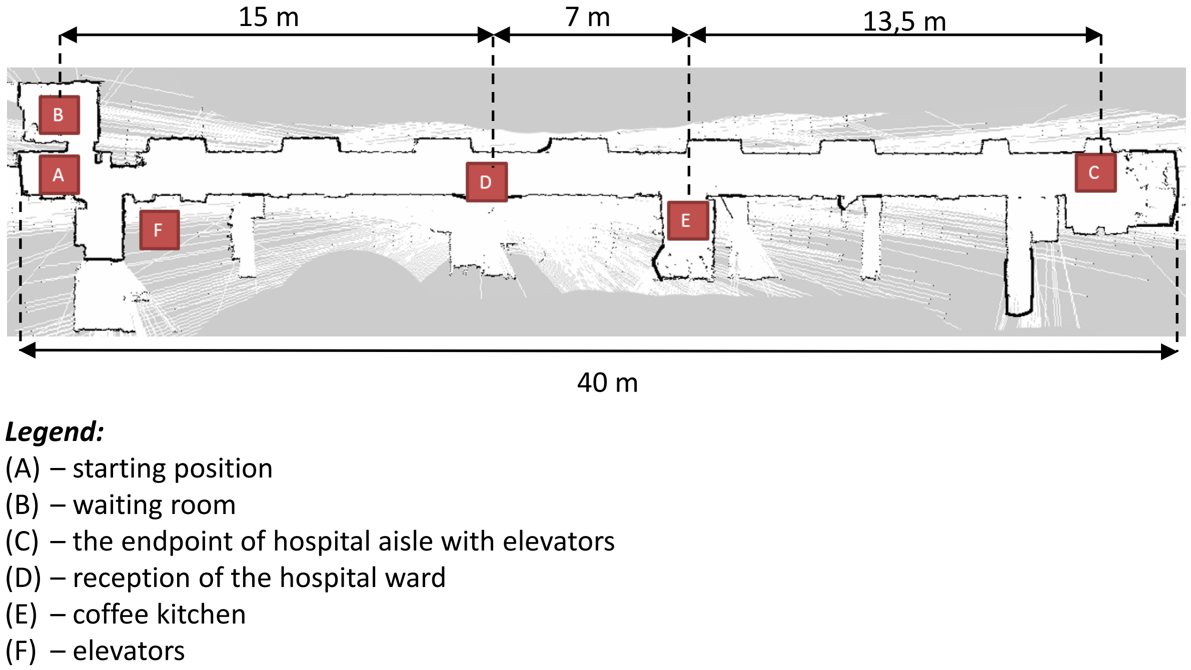


Figure 5.1.: Map of the Hospital Ward where the Mobile Service Robot has been evaluated in the iserveU Project (cf. [GRR$^+$16]).

In the context of iserveU, the behavior of the mobile service robot as well as that of the dynamic obstacles have been modeled as shown in Section 3.5.3 and respectively in Section 3.5.2, while the robot's safety property has been specified in TCTL as shown in Section 3.5.5. For the sake of convenience, the safety property formulated in Section

3.5.5 is reiterated in Equation (5.1). Notice that Equation (5.1) is the same as Equation (3.14) in which the notations from Notation 3.5.1 were used.

$$\begin{aligned}\phi : A\Box\ \{\mathit{forall}\ (i : \mathit{int}[0, n-1])\ (y_R == y_{O_i}) \wedge (x_{O_i} - x_R \geq -c_R) \\ \wedge (x_{O_i} - x_R \leq c_R) \rightarrow (v_R == 0)\}\end{aligned} \tag{5.1}$$

Furthermore, the environment assumption has also been specified in TCTL, as shown in Equation (5.2).

$$\psi : A\Box\ \{\mathit{forall}\ (i : \mathit{int}[0, n-1])\ (v^{O_i}_{\mathit{Max}} \leq v^{O}_{\mathit{MaxAssumed}})\} \tag{5.2}$$

Notice that the environment assumption considered in the iserveU project requires that the specific maximum velocity of the dynamic obstacles in the robot's environment does not exceed a predefined value (cf. Equation (5.2)). This environment assumption has been used in iserveU to compute a safety distance, also denoted as collision distance in Section 3.1, which the robot has to maintain towards the dynamic and stationary obstacles in its environment. The design-time verification has been carried out with the Uppaal model checker. In order to enable the verification of the robot's safety property during its operation, two runtime verification components were developed: (1) a runtime monitor for the environment assumption and a (2) test oracle for the robot's safety property.

The environment assumptions monitor has been formulated as in Equations (4.4) and (4.6) in Section 4.1.3. For the sake of convenience, the environment assumptions monitor formulated in Section 4.1.3 is reiterated in Equation (5.3).

$$M_\psi : G\ [v^{O}_{\mathit{MaxObserved}} \leq v^{O}_{\mathit{MaxAssumed}}] \tag{5.3}$$

where $v^{O}_{\mathit{MaxObserved}}(\tau) = \max_{i>0, t\in[0,\tau]}(v_{O_i}(t))$, $\forall \tau \in \mathbb{N}$.

The test oracle has been formulated for the robot's safety property as in Equation (3.21) in Section 3.7. For the sake of convenience, the test oracle formulated in Section 3.7 is reiterated Equation (5.4).

$$M_\phi : G\ [\neg((y_R == y_O) \wedge (y_R == y_O) \wedge (y_R == y_O)) \vee (v_R == 0)] \tag{5.4}$$

The environment assumptions monitor has been used to validate the environment assumption during the operation of the mobile service robot on the hospital ward, while the test oracle has been used to check whether the robot's compliance with its safety property, i.e., whether the robot actively caused a collision or not during its operation.

During its evaluation on the hospital ward, the mobile service robot was tasked to transport a package with materials from point (B) to point (E). Since its starting position was point (A) during the evaluation, the path followed by the mobile service robot was: $(A) \rightarrow (B) \rightarrow (D) \rightarrow (E)$. Notice that, the robot only passes by point (D) on its way to point (E), since point (E) is its final destination. Point (D) was nevertheless the point where dynamic obstacles started their movement towards point (A) or point (B), e.g. a nurse pushing a cart with medical materials. The evaluation of the mobile service robot in the context of the remote management and control use case took into consideration only the path from point (B) to point (E). Figure 5.2 shows the mobile service robot during its transport assignment on its way towards point (E). During the evaluation, personal employed in the project acted on a voluntary basis as stand-ins for nurses or visitors on the hospital ward project personal. This was done to avoid insurance issues and to avoid disturbing the hospital ward's daily activities.

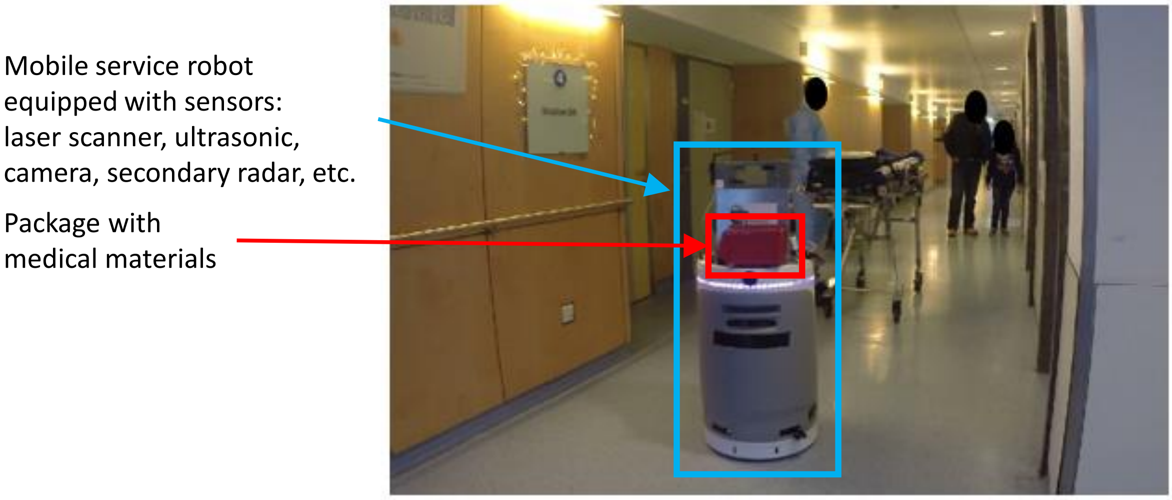

Mobile service robot equipped with sensors: laser scanner, ultrasonic, camera, secondary radar, etc.
Package with medical materials

Figure 5.2.: Mobile Service Robot en route from Point (B) to Point (E) on the Hospital Ward (cf. [GRR+16]).

Several test cases have been used to carry out the evaluation. Table 5.1 depicts the test cases used for the evaluation. Following inputs are considered for each test case:

- v^R_{Max} - specific maximum velocity of the robot,
- h_R - visual horizon of the robot's sensors,
- $v^O_{MaxAssumed}$ - maximum assumed obstacles velocity,
- $v^O_{MaxObserved}$ - maximum observed obstacles velocity, and
- v^O_{Max} - specific maximum velocity of dynamic obstacles.

During the execution of a test case, the robot's current velocity and the obstacle's current velocity vary in their respective value domains: $v_R \in [0, v^R_{Max}]$ and respectively $v_O \in [0, v^O_{Max}]$. The robot applied constant acceleration until it reached its v^R_{Max}. When it detected a collision danger or when it was approaching its destination point the robot started to brake in order to decrease its velocity. As for the dynamic obstacles, no restriction was placed on their acceleration or braking behavior. Instead, it was considered that a dynamic obstacle chooses randomly its current velocity v_O from the value domain $[0, v^O_{Max}]$. Notice that the maximum observed obstacle velocity $v^O_{MaxObserved}$ was equal to the specific maximum obstacle velocity v^O_{Max}, but this is not true in general. Consider as an example an automated vehicle moving on a highway, with a specific maximum velocity of 200 km/h, for which the maximum observed velocity is lower than 200 km/h. The test cases used during the evaluation on the hospital ward have been generated automatically via model checking during system test (cf. Section 3.7).

Table 5.1.: Mobile Service Robot: Test Cases for the Evaluation on the Hospital Ward.

ID	v^R_{Max}	h_R	$v^O_{MaxAssumed}$	$v^O_{MaxObserved}$	v^O_{Max}
TC1	0.2	1.5	0.2	0.2	0.2
TC2	0.2	1.5	0.2	0.3	0.3
TC3	0.2	1.5	0.2	0.25	0.25
TC4	0.2	1.6	0.2	0.3	0.3
TC5	0.2	1.4	0.2	0.3	0.3
TC6	0.2	1.4	0.2	0.25	0.25
TC7	0.2	1.3	0.2	0.3	0.3
TC8	0.2	1.2	0.2	0.3	0.3
TC9	0.2	1.2	0.2	0.25	0.25

Each test case has been executed ten times during the evaluation. Notice that, for a given test case, there were no two executions alike. This is because dynamic obstacles changed their velocity randomly, which in turn had an effect on the robot's behavior, e.g., the robot triggered the brake or changed to the side lane in order to avoid the obstacle. There were two questions posed for the evaluation in the iserveU project:

1. What effect does the (in)validity of environment assumptions have on the assessment results of the safety property during system operation time?
2. What effect does the size of the visual horizon of the robot's sensors have on the assessment results of the safety property during system operation time?

Figure 5.3 shows the evaluation results of the robot test cases that have been executed on the hospital ward in the iserveU project. Notice that in test case TC1 the environment assumption is valid, therefore no violations of the robot's safety property took place. In the test cases TC2 and TC3 the environment assumption has been invalidated by the maximum observed obstacle velocity, which led to violations of the robot's safety property taking place during the execution of these two test cases.

Compared to test case TC2, the test cases TC5 and TC7 decrease the visual horizon of the robot's sensors to 1.4 m and respectively 1.3 m. Although the number of safety property violations in TC5 is twice as large as the number of safety property violations in TC2, the decrease in the visual range of the robot's sensors from 1.4 m to 1.3 m in TC5 and TC7 does not influence the number of the robot's safety property violations in these test cases. The same can be said about test cases TC6 and TC9, where a change in the range of the robot sensors' horizon did ot change the number of safety property violations. This is not the case with test cases TC3 and TC6, where the number of safety

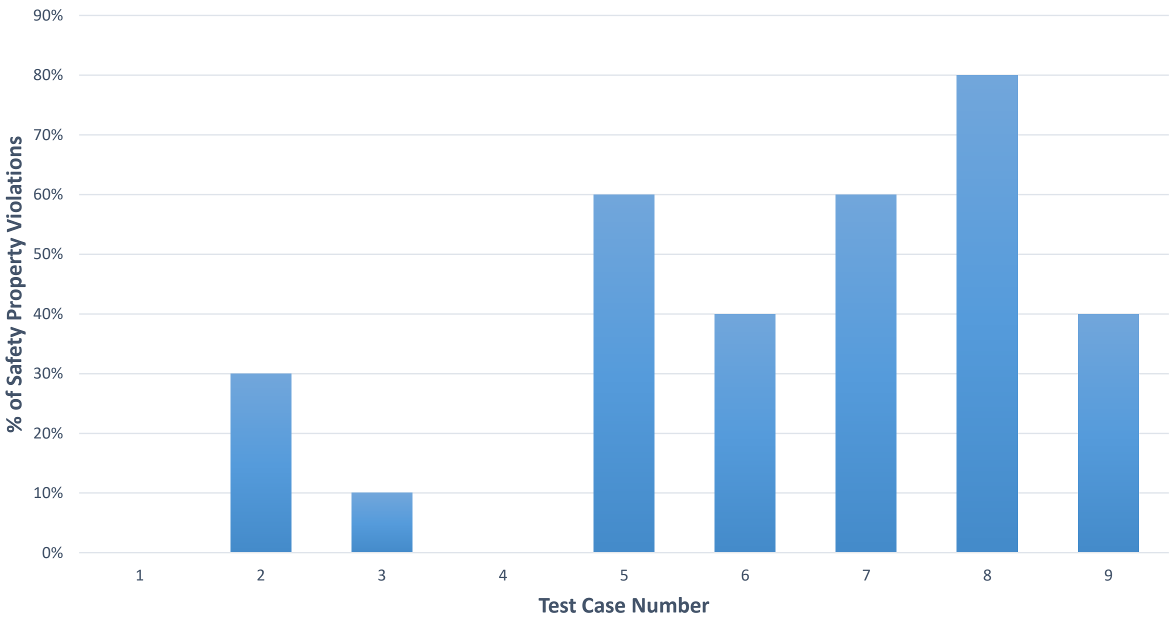


Figure 5.3.: Mobile Service Robot: Results of the Test Cases Evaluation on the Hospital Ward in the iserveU Project.

property violations in TC6 is four times as large as in TC3. Similarly, the number of safety property violations in TC5 and respectively TC7 is twice as much as the number of safety property violations in TC2.

The answer to question (1) of the evaluation is that invalid environment assumptions lead eventually to a violation of the robot's safety property. The effect of invalid environment assumptions can be counteracted if a larger upper limit for the robot's sensor horizon is chosen, as shown by test case TC4. With respect to question (2) of the evaluation, no definitive answer can be given, as the number of safety property violations stagnates in TC6 and TC9, and respectively in TC5 and TC7, or it increases drastically in case of test cases TC3 and TC6, and TC2 and TC5 or TC2 and TC7. A reason for this could be a cumulative effect between the invalid environment assumption and the respective decrease in the robot's sensor range. However, more tests need to be carried out in order to give a definitive answer to question (2) posed in the iserveU evaluation.

5.2. Case Study 2: Automotive System Function for Accurate Vehicle Speed Estimation

The second case study of this thesis is carried out on the basis of an automotive system function in charge of the accurate speed estimation of a driving vehicle. This section presents the application of this thesis' concept introduced in Chapter 4 to the vehicle speed estimation function. An abbreviated version of the vehicle speed estimation function has already been introduced in previous papers (cf. [AZR20, AVZR21]). Nevertheless, for the sake of completeness of this case study's presentation, details of the vehicle speed estimation function presented in [AZR20] and [AVZR21] are reiterated in this section. Section 5.2.1 presents an overview of the speed estimation function, while the following sections discuss the specific development phases carried for this function: (1) requirements elicitation and analysis (Section 5.2.2), (2) safety analysis (Section 5.2.3), (3) system design (Section 5.2.4), (4) system implementation (Section 5.2.5), (5) system test (Section 5.2.6), and (6) requirements validation (Section 5.2.7).

5.2.1. Overview of Vehicle Speed Estimation Function

Indicating the vehicle speed is a basic functionality introduced in vehicles at the beginning of the 20th century, which gains more attention in the context of advanced driving assistance systems and autonomous driving. This is done by a gauge installed in the vehicle which measures and displays the instantaneous speed of the car in miles per hour (mph), kilometer per hour (kph), or both (cf. [AZR20]). It is already possible from the perspective of the technical feasibility to obtain precise measurements of a vehicle speed by using high-quality reference measurement systems, e.g., inertial measurement unit (IMU) or differential GPS (D-GPS). The integration of such measurement systems in a series vehicle is however expensive for the automotive original equipment manufacturers (OEMs) and eventually cost-prohibitive for the end users. Therefore, the speed displayed on the instrument board of a commercialized series vehicle is not a precise measurement but an estimation of the actual vehicle speed performed with an accepted tolerance level and without using a reference measurement system (cf. [AVZR21]).

Recent vehicle speed estimation algorithms rely on measurements of the vehicle's wheel speed, which can be taken with onboard wheel speed sensors. In addition to the vehicle's wheel speed, these estimation algorithms also use the tire circumference for the computation of the vehicle speed. Depending on the current driving situation, the vehicle's tires are subject to a variety of deformations, due to the inherent elasticity of rubber. These deformations are also influenced by nondeterministic environmental factors, e.g., external temperature and road conditions, which make it impossible to obtain an exact value of the tire circumference of a driving vehicle (cf. [AVZR21]). Various approaches to estimate a vehicle speed have been patented throughout the years (cf. [Kos94],[DGRW04],[SHL17],[CIT20]).

The vehicle speed estimation function considered for this case study has been introduced in a previous paper (cf. [AZR20]). The function has been developed in the course of the research project *Accurate Vehicle Speed Estimation* (*Ger.:* Genaue Geschwindigkeitsermittlung), which was carried out in collaboration with an automotive OEM. The function uses GPS and wheel speed sensor data to carry out an estimation of the vehicle's tire circumference and compute the vehicle speed. The vehicle speed estimation function, denoted henceforth as VSE, has four major components or modules: (1) module M1 - estimation of the tire circumference, (2) module M2 - plausibility check of the tire circumference, (3) module M3 - computation of the vehicle speed, and (4) module M4 -

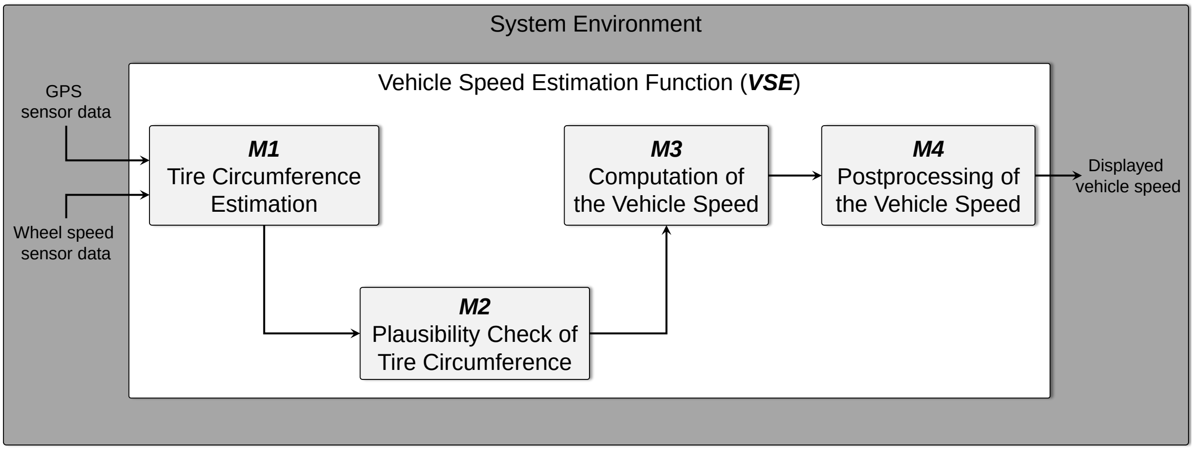

System Environment
Vehicle Speed Estimation Function (*VSE*)
GPS sensor data
Wheel speed sensor data
M1 Tire Circumference Estimation
M2 Plausibility Check of Tire Circumference
M3 Computation of the Vehicle Speed
M4 Postprocessing of the Vehicle Speed
Displayed vehicle speed

Figure 5.4.: Vehicle Speed Estimation Function: High-level Overview (cf. [AVZR21]).

post-processing of the vehicle speed curve, which includes rounding and smoothing (cf. [AZR20], [AVZR21]). An overview of the VSE function is given in Figure 5.4.

5.2.2. Requirements Elicitation and Analysis

The requirements for the VSE function are derived from compulsory regulations. Let $v_{Display}$ be the speed displayed on the dashboard of the ego-vehicle and v_{Real} be the actual vehicle speed. A directive of the Council of the European Economic Community imposed in 1975 the requirement in Equation (5.5) for the speed indicated by a car's speedometer (cf. [Eurb]):

$$0 \leq v_{Display} - v_{Real} \leq 0.1 \cdot v_{Real} + 4\ \frac{km}{h} \tag{5.5}$$

under the precondition that:

$$40 \leq v_{Real} \leq 120\ \frac{km}{h} \tag{5.6}$$

The proposed vehicle speed estimation algorithm must satisfy the requirement in Equation (5.7) introduced by the assessment protocol of the European New Car Assessment Programme in 2017 (cf. [Eura]).

$$0 \leq v_{Display} - v_{Real} \leq 5\ \frac{km}{h} \tag{5.7}$$

This requirement must be held under the assumption that the vehicle does not drive slower than 50 $\frac{km}{h}$ or faster than 120 $\frac{km}{h}$:

$$50 \leq v_{Real} \leq 120\ \frac{km}{h} \tag{5.8}$$

Notice that the lower bound of the deviation between $v_{Display}$ and v_{Real} is set to zero due to the safety reasons. This means that the vehicle speed indicated on the speedometer should never undercut the actual speed of the vehicle. The deviation between the displayed speed and the real speed of the vehicle plays an important role in further decisions which the driver or an assistance system may take during driving. Thus, in case the displayed speed is lower actual vehicle speed, it is possible that the driver or the driving assistance system might decide on a too conservative braking strategy or may not activate the brakes at all, which may lead to a collision actively caused by the ego-vehicle.

Besides implementation details of the VSE function, the OEM has also provided the target vehicle configuration in the course of the project. The target configuration has been used to derive further constraints, which define the physical limits of the tire circumference and the physical boundaries of the wheel speed measured by the wheel speed sensor. The minimum and maximum values of the tire circumference are denoted tc_{min} and tc_{max}, and represent its lower and upper physical limits. Notice that these physical boundaries are specific for each tire profile. The VSE algorithm assumes the following boundaries for tc_{Real}, the actual tire circumference:

$$\begin{aligned} &tc_{Min} \leq tc_{Real} \leq tc_{Max} \\ &tc_{Min} = 2118\ mm,\ tc_{Max} = 2293\ mm \end{aligned}$$

The physical boundaries of the tire circumference and the maximum vehicle speed error of 5 $\frac{km}{h}$ allowed by the NCAP requirement serve as a basis to derive the lower and upper physical limits, ω_{Min} and ω_{Max}, for the current wheel speed $\omega_{Current}$ measured by the wheel speed sensor:

$$\begin{aligned} &\omega_{Min} \leq \omega_{Current} \leq \omega_{Max} \\ &\omega_{Min} = 6.056\ \frac{1}{s},\ \omega_{Max} = 15.738\ \frac{1}{s} \end{aligned}$$

5.2.3. Safety Analysis

The HARA method is used for the safety analysis for the speed estimation function. For the hazard analysis, the following aspects along with the respective parameter values and equivalence classes are considered:

- the *location* where the vehicle operates:

$$Location := \{Outdoors\}$$

- the *geometry* of the physical world in which the vehicle moves, specified through two parameters, namely the number of lanes $n_{Lanes} \in \mathbb{N}_{>0}$, the road slope $\alpha_{Road} \in \mathbb{R}_{>0}$ and the curvature of the road $\gamma_{Road} \in \mathbb{R}_{\geq 0}$:

$$PhysicalWorldGeometry := \{(n_{Lanes} \geq 1,\ \alpha_{Road} \leq 12\%, \gamma_{Road} = 0°), \ldots\}$$

- the *driving conditions*:

$$DrivingConditions := \{(Dry, Non\text{-}slippery), (Wet, Slippery), \ldots\}$$

- the *environment* of the speed estimation function:

$$\begin{aligned}
&\mathit{Environment} &&:= \mathit{TechnicalEnvironment} \times \mathit{PhysicalEnvinronment}\\
&\mathit{TechnicalEnvironment} &&:= \{\mathit{CANBus}, \mathit{ESC}\}\\
&\mathit{PhysicalEnvironment} &&:= \{\mathit{UrbanArea}, \mathit{RuralArea}, \mathit{AlpineArea}, \dots\}
\end{aligned}$$

- the *function usage*, such as estimation of vehicle speed,
- the *system behavior* refers to the behavior of the vehicle in which the speed estimation function is operating:

$$\begin{aligned}
\mathit{SystemBehavior} := \{&\mathit{Accelerating}, \mathit{FullSpeedDrv}, \mathit{ReducedSpeedDrv},\\
&\mathit{Braking}, \mathit{Stopped}\}
\end{aligned}$$

- the *environment behavior*, which refers to the behavior exhibited in the technical environment and in the physical environment of the speed estimation function:

$$\begin{aligned}
&\mathit{EnvironmentBehavior} &&:= \mathit{TechnicalEnvBehavior} \times \mathit{PhysicalEnvBehavior}\\
&\mathit{TechnicalEnvBehavior} &&:= \{\mathit{ESC}, \dots\}\\
&\mathit{ESC} &&:= \{(s_{\mathit{VehicleRunning}} = \mathit{true}, a_{\mathit{Lat}} \leq 0.0001,\\
& &&\qquad a_{\mathit{Long}} \leq 0.001, \omega_{\mathit{Current}} \geq 0), \dots\}\\
&\mathit{PhysicalEnvBehavior} &&:= \{\mathit{GPS}, \dots\}\\
&\mathit{GPS} &&:= \{(s_{\mathit{GPS}} = \mathit{true}, e_{\mathit{GPS}} \leq 0.15), \dots\}
\end{aligned}$$

Notice that the environment of the speed estimation function is a combination of the technical environment in which the function operates and of the physical environment in which the ego vehicle drives. The technical environment of the speed estimation function consists of all the hardware and software components with which the speed estimation function interacts, e.g., electronic stability control (ESC). The behavior of these components is exhibited through their respective interfaces. In the case of the ESC component, the respective component interface provides information whether the vehicle is actually running or not through the boolean flag $s_{\mathit{VehicleRunning}}$, the lateral and longitudinal acceleration of the vehicle, a_{Lat} and a_{Long}, as well as the vehicle wheel speed $\omega_{\mathit{Current}}$.

The physical environment is situated outside of the vehicle in which the speed estimation function operates. The behavior in the physical environment is captured with specific vehicle sensors which provide relevant sensor input data to the speed estimation function. In this case, the GPS sensor provides information about whether a valid GPS signal is detected or not through the boolean flag s_{GPS} as well as the respective GPS data error e_{GPS}.

The values of the parameters defined in the HARA analysis are combined with each other creating unique concrete situations, each of these situations being depicted in one table row of Table 5.2. Notice that there is an uncountable number of situations which may appear during the driving of the ego vehicle. Therefore, neither the list of identified situations nor the list of parameters are considered to be complete. Notice that, depending on the vehicle manufacturer, the definition of the technical environment as well as that of the physical environment varies, since different vehicle manufacturer use different approaches for the speed estimation. Depending on the concept for the speed estimation algorithm, the vehicle manufacturer may use inputs from other functions or from different sensors. Furthermore, the presence of some parameters in the HARA analysis is dependent on other parameters. This is visible in the description of the physical environment behavior. Notice that the physical environment of the speed estimation function is also the physical environment in which the ego vehicle moves. Thus, with respect to the physical environment of the VSE function, it makes sense to talk about the driving conditions, e.g., wet and slippery road, or about the geometry of the physical environment, e.g., a road slope α_{Road} larger than 12%, since these have an impact on the estimation of the vehicle speed. With regard to its technical environment, the data provided by the ESC, e.g., vehicle wheel speed, is available only when the vehicle is running.

Table 5.2 illustrates the hazards identified through the HARA analysis carried out on the speed estimation function. Each row in the table shows a unique concrete situation as a combination of concrete parameter values considered in the HARA analysis. In H1, the vehicle drives with full speed on a straight road with dry and non-slippery surface and a road gradient smaller than 12% situated in a rural area. The ESC provides a longitudinal acceleration above 0.001, while the lateral acceleration is under 0.0001. Due to the high acceleration the vehicle speed may be overestimated, which leads to an false display of the vehicle speed on the instrument board of the vehicle. A similar situation is depicted in H2, except that instead of driving at full speed the vehicle is braking abruptly. As a result, a skidding effect occurs and the lateral acceleration is larger than 0.0001. This may lead to the overestimation of the vehicle speed and an inaccurate speed being displayed on the vehicle speedometer.

Table 5.2.: Vehicle Speed Estimation Function: Hazard Analysis and Risk Assessment.

ID	Location	Physical World Geometry	Driving Conditions	Environment	Function Usage	System Behavior	Environment Behavior	Hazard	Potential Effect
H1	*Outdoors*	$\alpha_{Road} \leq 12\%$, $\gamma_{Road} = 0°$	*Dry, Non-slippery*	*RuralArea*	*Estimate Vehicle Speed*	*FullSpeedDrv*	($s_{VehicleRunning} = true$, $a_{Long} > 0.001$, $a_{Lat} \leq 0.0001$), ($s_{GPS} = true$, $e_{GPS} \leq 0.15$)	Overest. of vehicle speed	False speed display
H2	*Outdoors*	$\alpha_{Road} \leq 12\%$, $\gamma_{Road} = 0°$	*Dry, Non-slippery*	*RuralArea*	*Estimate Vehicle Speed*	*Braking*	($s_{VehicleRunning} = true$, $a_{Long} \leq 0.001$, $a_{Lat} > 0.0001$), ($s_{GPS} = true$, $e_{GPS} \leq 0.15$)	Overest. of vehicle speed	False speed display
H3	*Outdoors*	$\alpha_{Road} \leq 12\%$, $\gamma_{Road} = 0°$	*Dry, Non-slippery*	*UrbanArea*	*Estimate Vehicle Speed*	*Accelerating*	($s_{VehicleRunning} = true$, $a_{Long} \leq 0.001$, $a_{Lat} \leq 0.0001$), ($s_{GPS} = true$, $e_{GPS} > 0.15$)	Overest. of vehicle speed	False speed display
H4	*Outdoors*	$\alpha_{Road} \leq 12\%$, $\gamma_{Road} = 0°$	*Dry, Non-slippery*	*UrbanArea*	*Estimate Vehicle Speed*	*Accelerating*	($s_{VehicleRunning} = true$, $a_{Long} \leq 0.001$, $a_{Lat} \leq 0.0001$), ($s_{GPS} = false$, $e_{GPS} = NaN$)	Overest. of vehicle speed	False speed display
H5	*Outdoors*	$\alpha_{Road} > 12\%$, $\gamma_{Road} > 0°$	*Dry, Non-slippery*	*AlpineArea*	*Estimate Vehicle Speed*	*Accelerating*	($s_{VehicleRunning} = true$, $a_{Long} \leq 0.001$, $a_{Lat} \leq 0.0001$), ($s_{GPS} = true$, $e_{GPS} \leq 0.15$)	Overest. of vehicle speed	False speed display
...	...	...	...	...	...	...	...	...	...

A different type of situation is presented in H3 and respectively H4, where the vehicle accelerates smoothly on a straight road with a dry non-slippery surface and road gradient smaller than 12% situated in an urban area. Due to the position of the GPS satellites and the large buildings in the urban area, the GPS device mounted on the ego vehicle may not acquire a valid GPS signal or it may receive GPS data which has an error larger than 0.15. In these situations, the hazard is that the vehicle speed is overestimated and a false speed is displayed on the vehicle's speedometer. In situation H5, the vehicle accelerates smoothly on a curved road with a road gradient larger than 12% situated in an alpine area. Although the vehicle drives in an alpine area, the GPS sensor of the vehicle receives valid GPS signals with a GPS data error less than 0.15. Nevertheless, the hazard is that the vehicle speed is overestimated due to the road slope which is larger than 12%. The potential effect of this hazard is that the speedometer displays an inaccurate speed estimation of the vehicle.

Due to the uncertainty inherent to the environment of the VSE function, the NCAP requirement is formulated as a probabilistic safety property. The NCAP requirement is transcribed into the following informal specification in natural language:

Probabilistic Safety Property (Informal Specification). With a probability of at least 0.95, the speed estimation function shall not underestimate the real vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.

Table 5.3 presents the safety requirement of the speed estimation function, which is extended with various environment assumptions by using the requirement pattern introduced in Figure 4.10 (cf. Section 4.4.3). The last column of the table also shows which of the hazards identified in the HARA analysis are covered by the respective extended safety requirements. Notice that the environment assumptions refer to the technical environment as well as the physical environment of the speed estimation function. It is also worth noting that there is no 1-to-1 relation between extended safety requirements and the respective hazards covered by them. Thus, the extended safety requirement ESR1 addresses the environment assumptions concerning the GPS data error and covers the hazards H3 and H4. The extended safety requirement ESR2 addresses the assumption regarding the geometry of the physical world in which the vehicle is moving, and thus, covers the hazard H5, while ESR3 and ESR4 address the environment assumptions referring to the longitudinal and lateral acceleration of the ego vehicle and cover the hazards H1 and H2 respectively.

The formal specification of the extended safety requirements of the VSE function is built using the Definition 4.4.7 along with Grammar 4.1 and Grammar 4.2 (cf. Section 4.4.5). The formal specification is depicted alongside the respective informal specification in Table 5.4.

Table 5.3.: Vehicle Speed Estimation Function: Safety Requirement extended with Environment Assumptions.

ID	Environment Assumption Clause	Requirement Main Clause	Covered Hazards
ESR1	If the vehicle receives a valid GPS signal that has an error less than 0.15 with a probability of at least 0.75,	then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	H3 and H4
ESR2	If the vehicle drives on a road that has road gradient less than 12% with a probability of at least 0.99,	then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	H5
ESR3	If the vehicle's longitudinal acceleration is less than 0.001 with a probability of at least 0.99,	then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	H1
ESR4	If the vehicle's lateral acceleration is less than 0.0001 with a probability of at least 0.99,	then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	H2

Table 5.4.: Vehicle Speed Estimation Function: Informal and Formal Specification of the Safety Requirement extended with Environment Assumptions.

ID	Informal Specification	Formal Specification
ESR1	If the vehicle receives a valid GPS signal that has an error less than 0.15 with a probability of at least 0.75, then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	$P_{\geq 0.75}(G\ [(s_{GPS} = \mathit{true}) \wedge (e_{GPS} \leq 0.15)]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$
ESR2	If the vehicle drives on a road that has road gradient less than 12% with a probability of at least 0.99, then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	$P_{\geq 0.99}G\ [\alpha_{Road} < 12\%] \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$
ESR3	If the vehicle's longitudinal acceleration is less than 0.001 with a probability of at least 0.99, then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	$P_{\geq 0.99}(G\ [a_{Long} \leq 0.001]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$
ESR4	If the vehicle's lateral acceleration is less than 0.0001 with a probability of at least 0.99, then with a probability of at least 0.95 the speed estimation function shall not underestimate the vehicle speed and it shall not overestimate it by more than 5 $\frac{km}{h}$.	$P_{\geq 0.99}(G\ [a_{Lat} \leq 0.0001]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$

5.2.4. System Design

During the system design phase, a functional abstraction is created for the VSE function represented in Figure 5.4. The functional abstraction uses mathematical functions that work on infinite domains. The construction of the functional abstraction follows the systematic approach depicted in Figure 5.5. This approach is carried for the VSE function as a whole as well as for each of the function's modules.

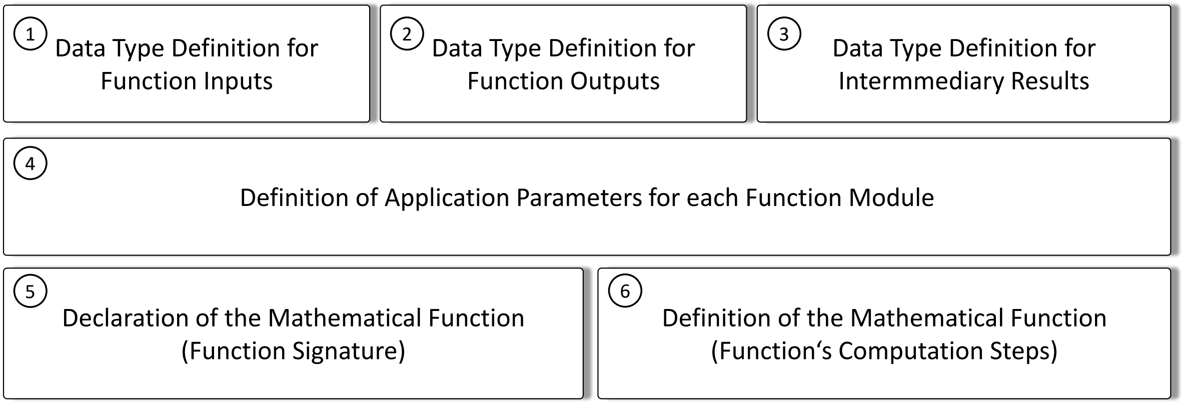

1 Data Type Definition for Function Inputs
2 Data Type Definition for Function Outputs
3 Data Type Definition for Intermmediary Results
4 Definition of Application Parameters for each Function Module
5 Declaration of the Mathematical Function (Function Signature)
6 Definition of the Mathematical Function (Function‘s Computation Steps)

Figure 5.5.: Vehicle Speed Estimation Function: Overview of Approach for Construction of the Functional Abstraction (cf. [AVZR21]).

Table 5.5 summarizes the data types for the input parameters and the output of the VSE function, while Table 5.6 displays the data types of the intermediary results obtained in each of the modules M1 to M4. Table 5.5 and Table 5.6 specify for each defined data type (1) the name of the data type, (2) the corresponding mathematical set, (3) the parameter category which is associated with the respective data type, (4) the mathematical function in which the respective parameter is used, and (5) an informal description of the data type. As an example, consider the data type $TSpeedGPS$, defined as equivalent to the set of real positive numbers $\mathbb{R}_{\geq 0}$ for the GPS speed of the ego vehicle, which is used as input parameter in the VSE function.

The application parameters used in the VSE function are defined in Table 5.7. For each application parameter, Table 5.7 provides (1) the parameter name, (2) the parameter value, (3) the parameter data type, (4) the mathematical function in which the application parameter is used, and (5) an informal description of the parameter. Notice that some of the application parameters defined in Table 5.7 are of data types introduced in Table 5.5, e.g., $e^{GPS}_{MaxAssumed} \in TErrorGPS$.

It is worth noticing that the defined data types build an abstract data type system, with operations that abide by mathematical laws independently of how the concrete data types are implemented in the automotive function and represented on the target control unit (cf. [AVZR21]).

Table 5.5.: Vehicle Speed Estimation Function - Definition of Data Types for the Function Inputs and the Function Outputs in the Functional Abstraction.

Data Type Name	**Corresponding Mathematical Set**	**Parameter Category**	**Mathematical Function**	**Data Type Description**
TSpeedGPS	$\mathbb{R}_{\geq 0}$	Input Parameter	VSE	Data type for the GPS speed of the ego vehicle
TErrorGPS	$\mathbb{R}_{\geq 0}$	Input Parameter	VSE	Data type for the GPS error
TLongAccel	$\mathbb{R}$	Input Parameter	VSE	Data type for the longitudinal acceleration of the ego vehicle
TLatAccel	$\mathbb{R}$	Input Parameter	VSE	Data type for the latitudinal acceleration of the ego vehicle
TRoadSlope	$\mathbb{R}$	Input Parameter	VSE	Data type for the angle of the road inclination
TWheelSpeed	$\mathbb{R}_{\geq 0}$	Input Parameter	VSE	Data type for the wheel speed of the ego vehicle
TVehicleRunning	*Bool*	Input Parameter	M2	Data type for the flag indicating whether the ego vehicle is running or not
TIndex	$\mathbb{N}$	Input Parameter	**tcErrorEstimation**	Data type for the index indicating the current iteration in the estimation of the tire circumference error
TDisplaySpeed	$\mathbb{R}_{\geq 0}$	Output	VSE	Data type for the ego vehicle speed displayed on the vehicle's instrument board
TDisplaySpeed	$\mathbb{R}_{\geq 0}$	Output	VSE	Data type for the ego vehicle speed displayed on the vehicle's instrument board

Table 5.6.: Vehicle Speed Estimation Function - Definition of Data Types for the Intermediary Results in the Functional Abstraction.

Data Type Name	**Corresponding Mathematical Set**	**Parameter Category**	**Mathematical Function**	**Data Type Description**
TEstimatedSpeed	$\mathbb{R}_{\geq 0}$	Intermediary Result	M3	Data type for the estimated vehicle speed without postprocessing
TPlausibleTC	$\mathbb{R}_{\geq 0}$	Intermediary Result	M2	Data type for the tire circumference obtained after the plausibility check
TSlopeLimitedTC	$\mathbb{R}_{\geq 0}$	Intermediary Result	M2	Data type for the tire circumference obtained after applying the slope limiting filter
TEstimatedTC	$\mathbb{R}_{\geq 0}$	Intermediary Result	M1	Data type for the estimated tire circumference
TErrorEstimatedTC	$\mathbb{R}_{\geq 0}$	Intermediary Result	M1	Data type for the estimated error of the tire circumference
TStandardTC	$\mathbb{R}_{\geq 0}$	Intermediary Result	M1	Data type for the standard tire circumference
TErrorUpdateTC	$\mathbb{R}$	Intermediary Result	M1	Data type for the error update of the tire circumference
TValidGPSError	*Bool*	Intermediary Result	**gpsErrorValid**	Data type for the flag indicating whether the GPS data is valid or not
TValidRoadSlope	*Bool*	Intermediary Result	**roadSlopeValid**	Data type for the flag indicating whether the road slope is below the admitted threshold or not
TValidLongAccel	*Bool*	Intermediary Result	**longAccelValid**	Data type for the flag indicating whether the longitudinal acceleration of the ego vehicle is valid or not
TValidLatAccel	*Bool*	Intermediary Result	**latAccelValid**	Data type for the flag indicating whether the lateral acceleration of the ego vehicle is valid or not

Table 5.7.: Vehicle Speed Estimation Function - Definition of Application Parameters.

Parameter Name	**Parameter Value**	**Measurement Unit**	**Parameter Data Type**	**Mathematical Function**	**Parameter Description**
a	-0.5152	-	$\mathbb{R}$	M1	First coefficient in the polynomial approximating the standard tire circumference
b	0.07646	-	$\mathbb{R}$	M1	Second coefficient in the polynomial approximating the standard tire circumference
c	2175	-	$\mathbb{R}_{\geq 0}$	M1	Third coefficient in the polynomial approximating the standard tire circumference
w	0.1	-	$\mathbb{R}_{\geq 0} \cap [0, 1]$	M1	Weighting factor for the estimated error of the tire circumference
$a_{MaxAssumed}^{Long}$	0.001	$\frac{m}{s^2}$	$TLongAccel$	M1	Upper bound assumed for the longitudinal acceleration of the vehicle
$a_{MaxAssumed}^{Lat}$	0.0001	$\frac{m}{s^2}$	$TLatAccel$	M1	Upper bound assumed for the lateral acceleration of the vehicle
$\alpha_{MaxAssumed}^{Road}$	12%	-	$TRoadSlope$	M1	Upper bound assumed for the angle of the road inclination
$e_{MaxAssumed}^{GPS}$	0.15	$\frac{m}{s}$	$TErrorGPS$	M1	Upper bound assumed for the GPS data error
tc_{Min}	2118	mm	$TPlausibleTC$	M2	Lower bound assumed for the tire circumference
tc_{Max}	2293	mm	$TPlausibleTC$	M2	Upper bound assumed for the tire circumference
P	20	$\frac{mm}{3*s}$	$\mathbb{R}_{\geq 0}$	M2	Slope limitation for the estimated tire circumference, corresponds to 20 mm in 3 s.

The declaration of a mathematical function in the functional abstraction is in fact the signature of the mathematical function. The signature of a mathematical function consists of the function's name, the data types and names of the function's input parameters and the data type of the function's return value. The definition of the mathematical function is the function's block in which the computation steps of the function are described. The definition of the mathematical function is formulated through a *walk-back* from the function's output towards the function's inputs, in a similar manner to the notation in functional programming (cf. [AVZR21]).

The *walk-back* approach applies for the definition of any mathematical function in the functional abstraction, independent of the level at which this function is defined in the hierarchy of the functional abstraction. This is best visible in the definition of the mathematical function VSE in Equation (5.11). In the following sections, the functionality of the modules M1 to M4 is explained in more detail using the definitions of the respective mathematical functions.

$$\begin{aligned}\mathbf{VSE} :\,& \mathit{TVehicleRunning} \times \mathit{TSpeedGPS} \times \mathit{TErrorGPS} \times \mathit{TLongAccel} \times \\ & \mathit{TLatAccel} \times \mathit{TRoadSlope} \times \mathit{TWheelSpeed} \rightarrow \mathit{TDisplaySpeed}\end{aligned} \tag{5.9}$$

$$\begin{aligned}\mathit{TDisplaySpeed}\ \mathbf{VSE}\ (&\mathit{TVehicleRunning}\ s_{\mathit{VehicleRunning}}, \\ &\mathit{TSpeedGPS}\ v_{\mathit{GPS}}, \mathit{TErrorGPS}\ e_{\mathit{GPS}}, \\ &\mathit{TLongAccel}\ a_{\mathit{Long}}, \mathit{TLatAccel}\ a_{\mathit{Lat}}, \\ &\mathit{TRoadSlope}\ \alpha_{\mathit{Road}}, \mathit{TWheelSpeed}\ \omega_{\mathit{Current}})\end{aligned} \tag{5.10}$$

$$\begin{aligned}&\mathbf{VSE}\ (s_{\mathit{VehicleRunning}}, v_{\mathit{GPS}}, e_{\mathit{GPS}}, a_{\mathit{Long}}, a_{\mathit{Lat}}, \alpha_{\mathit{Road}}, \omega_{\mathit{Current}}) = v_{\mathit{Display}} \\ &v_{\mathit{Display}} = \mathit{M4}\ (v_{\mathit{Estimated}}) \\ &v_{\mathit{Estimated}} = \mathit{M3}\ (\omega_{\mathit{Current}}, tc_{\mathit{Plausible}}) \\ &tc_{\mathit{Plausible}} = \mathit{M2}\ (s_{\mathit{VehicleRunning}}, tc_{\mathit{Estimated}}) \\ &tc_{\mathit{Estimated}} = \mathit{M1}\ (v_{\mathit{GPS}}, e_{\mathit{GPS}}, a_{\mathit{Long}}, a_{\mathit{Lat}}, \alpha_{\mathit{Road}}, \omega_{\mathit{Current}})\end{aligned} \tag{5.11}$$

Estimation of the Tire Circumference

Module M1 computes the estimation of the vehicle's tire circumference, which is modeled by the respective mathematical function in Equations (5.12) to (5.14). As shown in Equation (5.14), the estimated tire circumference has two components: the standard tire circumference and the tire circumference error. The first component is an approximation of the standard tire circumference based on the measurements of the vehicle's wheel speed, while the second component is an estimation of the tire circumference error on the basis of selected GPS sensor data (cf. [AZR20]).

The standard tire circumference tc_{Standard} is approximated based on the measurements of the ego vehicle's wheel speed $\omega_{\mathit{Current}}$, according to the polynomial defined for tc_{Standard} in Equation (5.14). The application parameters a, b, and c used in the polynomial of tc_{Standard} are defined in Table 5.7.

$$\begin{aligned}\textbf{M1}:&\mathit{TSpeedGPS} \times \mathit{TErrorGPS} \times \mathit{TLongAccel} \times \mathit{TLatAccel} \times \\ &\mathit{TRoadSlope} \times \mathit{TWheelSpeed} \rightarrow \mathit{TEstimatedTC}\end{aligned} \tag{5.12}$$

$$\begin{aligned}\mathit{TEstimatedTC}\ \textbf{M1}\ (&\mathit{TSpeedGPS}\ v_{\mathit{GPS}}, \mathit{TErrorGPS}\ e_{\mathit{GPS}}, \mathit{TLongAccel}\ a_{\mathit{Long}}, \\ &\mathit{TLatAccel}\ a_{\mathit{Lat}}, \mathit{TRoadSlope}\ \alpha_{\mathit{Road}}, \mathit{TWheelSpeed}\ \omega_{\mathit{Current}})\end{aligned} \tag{5.13}$$

$$\begin{aligned}&\textbf{M1}\ (v_{\mathit{GPS}}, e_{\mathit{GPS}}, a_{\mathit{Long}}, a_{\mathit{Lat}}, \alpha_{\mathit{Road}}, \omega_{\mathit{Current}}) = tc_{\mathit{Estimated}} \\ &tc_{\mathit{Estimated}} = \Delta tc_{\mathit{Estimated}} + tc_{\mathit{Standard}} \\ &\Delta tc_{\mathit{Estimated}} = \mathit{tcErrorEstimation}\ (v_{\mathit{GPS}}, e_{\mathit{GPS}}, a_{\mathit{Long}}, a_{\mathit{Lat}}, \\ &\qquad\qquad \alpha_{\mathit{Road}}, \omega_{\mathit{Current}}, i, tc_{\mathit{Standard}}) \\ &tc_{\mathit{Standard}} = a * \sin(\omega_{\mathit{Current}} - \pi) + b * (\omega_{\mathit{Current}} - 10)^2 + c\end{aligned} \tag{5.14}$$

$$\begin{aligned}\textbf{tcErrorEstimation}:&\mathit{TSpeedGPS} \times \mathit{TErrorGPS} \times \mathit{TLongAccel} \times \mathit{TLatAccel} \times \\ &\mathit{TRoadSlope} \times \mathit{TWheelSpeed} \times \mathit{TIndex} \times \mathit{TStandardTC} \\ &\rightarrow \mathit{TErrorEstimatedTC}\end{aligned} \tag{5.15}$$

$$\begin{aligned}\mathit{TErrorEstimatedTC}\ \textbf{tcErrorEstimation}\ (&\mathit{TSpeedGPS}\ v_{\mathit{GPS}}, \mathit{TErrorGPS}\ e_{\mathit{GPS}}, \\ &\mathit{TLongAccel}\ a_{\mathit{Long}}, \mathit{TLatAccel}\ a_{\mathit{Lat}}, \\ &\mathit{TRoadSlope}\ \alpha_{\mathit{Road}}, \mathit{TWheelSpeed}\ \omega_{\mathit{Current}}, \\ &\mathit{TIndex}\ i, \mathit{TStandardTC}\ \Delta tc_{\mathit{Standard}})\end{aligned} \tag{5.16}$$

$$\begin{aligned}&\textbf{tcErrorEstimation}\ (v_{\mathit{GPS}}, e_{\mathit{GPS}}, a_{\mathit{Long}}, a_{\mathit{Lat}}, \alpha_{\mathit{Road}}, \omega_{\mathit{Current}}, i, tc_{\mathit{Standard}}) \\ &= \Delta tc_{\mathit{Estimated}} \\ &\Delta tc_{\mathit{Estimated}} = \begin{cases}\Delta tc_{\mathit{Init}}, i = 0 \\ (1-w) * \Delta tc_{\mathit{Estimated}} + w * \Delta tc_{\mathit{Update}}, i > 0 \wedge \mathit{updateFlag}\end{cases} \\ &\Delta tc_{\mathit{Init}} = \frac{2.5}{\omega_{\mathit{Current}}} * \frac{1000}{3.6} \\ &\Delta tc_{\mathit{Update}} = \mathit{tcErrorUpdate}\ (tc_{\mathit{Standard}}, v_{\mathit{GPS}}, \omega_{\mathit{Current}}) \\ &\mathit{updateFlag} = \mathit{gpsErrorValid}\ (e_{\mathit{GPS}}) \wedge \mathit{roadSlopeValid}\ (\alpha_{\mathit{Road}}) \wedge \\ &\qquad \mathit{longAccelValid}\ (a_{\mathit{Long}}) \wedge \mathit{latAccelValid}\ (a_{\mathit{Lat}})\end{aligned} \tag{5.17}$$

The tire circumference error $\Delta tc_{\mathit{Estimated}}$ is estimated on the basis of the GPS sensor data, which consists of measurements of the GPS vehicle speed as well as information about the quality of the received data. The quality of the GPS data is expressed by a GPS error factor. A strong GPS signal means a high quality of the received GPS data, which in turn translates in low GPS error factor (cf. [AZR20]). In the first iteration, the tire circumference error $\Delta tc_{\mathit{Estimated}}$ is initialized with the value of $\Delta tc_{\mathit{Init}}$. The subsequent iterations update the tire circumference error $\Delta tc_{Estimated}$ on-the-fly based on the current GPS measurements. Thus, the estimated error of the tire circumference is a function of the previous estimated value of this error and an update based on current GPS sensor data, weighted by the weighting factor w defined as an application parameter in Table 5.7 (cf. [AZR20]). The update of the tire circumference error occurs only when the *updateFlag* in the *tcErrorEstimation* function evaluates to the truth value *true*. This is achieved only if the information received by the VSE function from the technical and physical environment is valid. From the technical environment, the function receives the longitudinal acceleration a_{Long} and the lateral acceleration a_{Lat}. From the physical environment, the speed estimation function receives inputs regarding the GPS data error e_{GPS} and the road slope α_{Road}. The validity of the received information is checked in the functional abstraction of the speed estimation function with the help of the first-order predicates *gpsErrorValid*, *roadSlopeValid*, *longAccelValid*, and *latAccelValid*, whose respective mathematical functions are declared and defined in Equations (5.18) to (5.21).

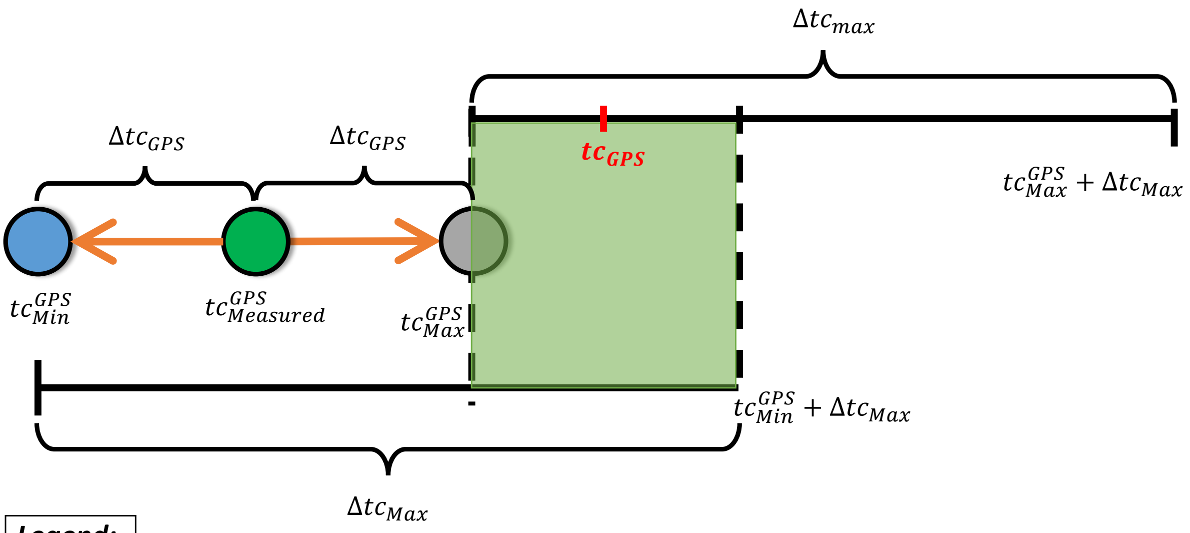
$$\begin{aligned}
&\mathbf{gpsErrorValid} : \mathit{TErrorGPS} \rightarrow \mathit{TValidGPSError}\\
&\mathit{TValidGPSError}\ \mathbf{gpsErrorValid}\ (\mathit{TErrorGPS}\ e_{GPS})\\
&\mathbf{gpsErrorValid}\ (e_{GPS}) = \mathit{flagGPSError}\\
&\mathit{flagGPSError} = (s_{GPS} == \mathit{true}) \wedge (e_{GPS} \leq e^{GPS}_{MaxAssumed})
\end{aligned} \tag{5.18}$$

$$\begin{aligned}
&\mathbf{roadSlopeValid} : \mathit{TRoadSlope} \rightarrow \mathit{TValidRoadSlope}\\
&\mathit{TValidRoadSlope}\ \mathbf{roadSlopeValid}\ (\mathit{TRoadSlope}\ \alpha_{Road})\\
&\mathbf{roadSlopeValid}\ (\alpha_{Road}) = \mathit{flagRoadSlope}\\
&\mathit{flagRoadSlope} = (\alpha_{Road} \leq \alpha^{Road}_{MaxAssumed})
\end{aligned} \tag{5.19}$$

$$\begin{aligned}
&\mathbf{longAccelValid} : \mathit{TLongAccel} \rightarrow \mathit{TValidLongAccel}\\
&\mathit{TValidLongAccel}\ \mathbf{longAccelValid}\ (\mathit{TLongAccel}\ a_{Long})\\
&\mathbf{longAccelValid}\ (a_{Long}) = \mathit{flagLongAccel}\\
&\mathit{flagLongAccel} = (a_{Long} \leq a^{Long}_{MaxAssumed})
\end{aligned} \tag{5.20}$$

$$\begin{aligned}
&\mathbf{latAccelValid} : \mathit{TLatAccel} \rightarrow \mathit{TValidLatAccel}\\
&\mathit{TValidLatAccel}\ \mathbf{latAccelValid}\ (\mathit{TLatAccel}\ a_{Lat})\\
&\mathbf{latAccelValid}\ (a_{Lat}) = \mathit{flagLatAccel}\\
&\mathit{flagLatAccel} = (a_{Lat} \leq a^{Lat}_{MaxAssumed})
\end{aligned} \tag{5.21}$$

In order to comply with the Euro NCAP requirement, the update to the tire circumference error is computed with adequate GPS data that satisfy the environment assumption with respect to the GPS error. Any other GPS data that does not fulfill the environment assumption are discarded. The mathematical function *tcErrorUpdate*, whose declaration and definition is given in Equations (5.22) to (5.24), specifies a mechanism



Legend:	
$tc_{Measured}^{GPS}$ – measured GPS tire circumference	Δtc_{GPS} – tire circumference error corresponding to the maximum positive GPS error
tc_{Min}^{GPS} – GPS tire circumference with minimum negative GPS error	Δtc_{Max} – maximum tire circumference error allowed by the Euro NCAP requirement
tc_{Max}^{GPS} – GPS tire circumference with maximum positive GPS error	tc_{GPS} - good value of the GPS measured tire circumference chosen to compute the error update of the estimated tire circumference

Figure 5.6.: Vehicle Speed Estimation Function: Visual Intuition of the Mechanism for Adequate GPS Data Selection (cf. [AZR20]).

by which intervals of adequate GPS data are identified and selected from the whole data set received by the vehicle's GPS sensor. A visual intuition of this mechanism is depicted in Figure 5.6.

$$\begin{aligned}\textbf{tcErrorUpdate}:\,& \mathit{TStandardTC} \times \mathit{TSpeedGPS} \times \\ & \mathit{TWheelSpeed} \rightarrow \mathit{TErrorUpdateTC}\end{aligned} \tag{5.22}$$

$$\begin{aligned}\mathit{TErrorUpdateTC}\ \textbf{tcErrorUpdate}\ (&\mathit{TStandardTC}\ tc_{\mathit{Standard}}, \mathit{TSpeedGPS}\ v_{GPS}, \\ &\mathit{TWheelSpeed}\ \omega_{\mathit{Current}})\end{aligned} \tag{5.23}$$

$$\begin{aligned}
&\textbf{tcErrorUpdate}(tc_{\mathit{Standard}}, v_{GPS}, \omega_{\mathit{Current}}) = \Delta tc_{\mathit{Update}} \\
&\Delta tc_{\mathit{Update}} = tc_{GPS} - tc_{\mathit{Standard}} \\
&tc_{GPS} = \frac{tc^{GPS}_{\mathit{Max}} + (tc^{GPS}_{\mathit{Min}} + \Delta tc_{\mathit{Max}})}{2} \\
&tc^{GPS}_{\mathit{Max}} = tc^{GPS}_{\mathit{Measured}} + \Delta tc_{GPS} \\
&tc^{GPS}_{\mathit{Min}} = tc^{GPS}_{\mathit{Measured}} - \Delta tc_{GPS} \\
&tc^{GPS}_{\mathit{Measured}} = \frac{v_{GPS} * 1000}{\omega_{\mathit{Current}}} \\
&\Delta tc_{GPS} = \frac{e^{GPS}_{\mathit{MaxAssumed}} * 1000}{\omega_{\mathit{Current}}} \\
&\Delta tc_{\mathit{Max}} = \frac{5}{\omega_{\mathit{Current}}} * \frac{1000}{3.6}
\end{aligned} \tag{5.24}$$

Plausibility Check of the Tire Circumference

Each tire profile specifies lower and upper limits for the tire circumference, which represent the physical boundaries of the real and of the estimated tire circumferences. In order to have an estimation of the vehicle speed as accurate as possible, plausibility checks are carried out in order to eliminate any outliers (cf. [AZR20]).

Module M2, whose mathematical function is declared and defined in Equations (5.25) to (5.27), filters out the physically impossible values of the estimated tire circumference by using the physical boundaries defined for the tire circumference in Table 5.7. Notice that before the plausibility check is performed, the estimated tire circumference is filtered with a slope limiting filter, which ensures that the difference between the current value and the old value of the estimated tire circumference does not exceed the threshold of 20 mm in 3 s defined by the application parameter P (cf. Table 5.7). This is necessary in order to ensure that only relevant data is used for the estimation of the tire circumference (cf. [AZR20]).

$$\mathit{M2} : \mathit{TVehicleRunning} \times \mathit{TEstimatedTC} \rightarrow \mathit{TPlausibleTC} \tag{5.25}$$

$$\mathit{TPlausibleTC}\ \mathit{M2}\ (\mathit{TVehicleRunning}\ s_{\mathit{VehicleRunning}}, \mathit{TEstimatedTC}\ tc_{\mathit{Estimated}}) \tag{5.26}$$

$$\begin{aligned}
&\mathit{M2}\ (s_{\mathit{VehicleRunning}}, tc_{\mathit{Estimated}}) = tc_{\mathit{Plausible}} \\
&tc_{\mathit{Plausible}} = max(TC_{\mathit{Min}}, min(tc_{\mathit{SlopeLimited}}, TC_{\mathit{Max}})) \\
&tc_{\mathit{SlopeLimited}} = \begin{cases} \mathit{slopeLimitation}(pre(tc_{\mathit{Plausible}}), tc_{\mathit{Estimated}}), & s_{\mathit{VehicleRunning}} = \mathit{true} \\ tc_{\mathit{Estimated}}, & s_{\mathit{VehicleRunning}} = \mathit{false} \end{cases}
\end{aligned} \tag{5.27}$$

$$\mathit{slopeLimitation} : \mathit{TPlausibleTC} \times \mathit{TEstimatedTC} \rightarrow \mathit{TSlopeLimitedTC} \tag{5.28}$$

$$\begin{aligned}
\mathit{TSlopeLimitedTC}\ \mathit{slopeLimitation}(\ &\mathit{TPlausibleTC}\ tc^{\mathit{Old}}_{\mathit{Plausible}}, \\
&\mathit{TEstimatedTC}\ tc^{\mathit{New}}_{\mathit{Estimated}})
\end{aligned} \tag{5.29}$$

$$\begin{aligned}
&\mathit{slopeLimitation}(tc^{\mathit{Old}}_{\mathit{Plausible}}, tc^{\mathit{New}}_{\mathit{Estimated}}) = tc_{\mathit{SlopeLimited}} \\
&tc_{\mathit{SlopeLimited}} = \begin{cases} tc^{\mathit{Old}}_{\mathit{Plausible}} + sgn(\Delta) * P, & |\Delta| > P \\ tc^{\mathit{New}}_{\mathit{Estimated}}, & |\Delta| \leq P \end{cases} \\
&\Delta = tc^{\mathit{New}}_{\mathit{Estimated}} - tc^{\mathit{Old}}_{\mathit{Plausible}}
\end{aligned} \tag{5.30}$$

Notation 5.2.1. *pre* is an operator used in Equation (5.27) to obtain the value of a given parameter computed in a previous iteration. The *pre* operator is defined as follows:

$$\begin{aligned}
&pre : \mathbb{R} \times \mathbb{N} \rightarrow \mathbb{R} \\
&pre(x, k) \overset{\text{def}}{=} \begin{cases} x, & k = 0 \\ pre(x), & k = 1 \\ \underbrace{pre(pre(pre(\ldots pre}_{\text{k}}(x)))), & k > 1 \end{cases}
\end{aligned}$$

sgn denotes the sign function and it is used in Equation (5.30) to determine the sign of the difference between the new tire circumference value estimated in the current execution step and the old tire circumference value computed through the plausibility check in the previous execution step. The *sgn* function is defined as follows:

$$\begin{aligned}
&sgn : \mathbb{R} \rightarrow \{-1, 0, 1\} \\
&sgn(x) \overset{\text{def}}{=} \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}
\end{aligned}$$

Should the difference between the two values of the estimated tire circumference be larger than the margin of 20 *mm* in 3 *s*, then it means that data necessary for the estimation of the tire circumference has been missing, e.g., when the vehicle is not running. In such a case, the old value of the estimated tire circumference is discarded since it is not useful anymore. In such a case, the computation continues with the new estimated tire circumference (cf. [AZR20]). This is reflected by the mathematical function *slopeLimitation* declared and defined in Equations (5.28) to (5.30).

Computation of the Vehicle Speed

Module M3, modeled by Equations (5.31) to (5.33), computes the instantaneous speed of the vehicle using the plausible tire circumference and the current wheel speed measured by the vehicle's wheel speed sensors.

$$M3 : TWheelSpeed \times TPlausibleTC \rightarrow TEstimatedSpeed \tag{5.31}$$

$$TEstimatedSpeed\ M3\ (TWheelSpeed\ \omega_{Current}, TPlausibleTC\ tc_{Plausible}) \tag{5.32}$$

$$\begin{aligned} &M3\ (\omega_{Current}, tc_{Plausible}) = v_{Estimated} \\ &v_{Estimated} = \frac{3.6}{1000} * \omega_{Current} * tc_{Plausible} \end{aligned} \tag{5.33}$$

Postprocessing of the Vehicle Speed

The speed displayed on instrument board of the ego vehicle is computed from the estimated vehicle speed by rounding it off so that it matches the speedometer's value range. Since sudden moves of the pointer needle on the speedometer may be confusing for the driver, a smoothing function is applied to the resulting curve of the vehicle speed in order to avoid the needle bouncing back and forth at every small change in the estimated vehicle speed (cf. [AZR20]).

$$M4 : TEstimatedSpeed \rightarrow TDisplaySpeed \tag{5.34}$$

$$TDisplaySpeed\ M4\ (TEstimatedSpeed\ v_{Estimated}) \tag{5.35}$$

$$\begin{aligned} &M4\ (v_{Estimated}) = v_{Display} \\ &v_{Display} = smoothSpeedCurve\ (v_{Rounded}) \\ &v_{Rounded} = roundVehicleSpeed\ (v_{Estimated}) \end{aligned} \tag{5.36}$$

Design-Time Verification

For the design-time verification, the functional abstraction of the VSE function is modeled as an MDP in the Prism modeling language. The design-time verification is shown

Table 5.8.: Vehicle Speed Estimation Function: Verification of Extended Safety Requirement ESR1 expressed as Multi-objective Property.

ID	Formal Specification	Multi-objective Property	Verification Result
ESR1	$P_{\geq 0.75}(G\ [(s_{GPS} == \mathit{true}) \land (e_{GPS} \leq 0.15)]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \land (v_{Display} - v_{Real} \leq 5)])$	`multi(P>= 0.75 [G ((sGPS = true) & (eGPS <= 0.15))], P>=0.95 [G ((0 <= vDisplay - vReal) & (vDisplay - vReal <= 5))])`	✓

in this section exemplary on the basis of the extended safety requirement ESR1. The extended safety requirement ESR1, formally specified in Table 5.4, is translated as a multi-objective property in PCTL and verified with the Storm model checker. Table 5.8 shows the multi-objective property alongside the formal specification of ESR1, together with its verification result. The translation to multi-objective properties of the other three extended safety requirements specified in Table 5.4 is shown together with the respective design-time verification results in Appendix A.

Definition of Environment Assumptions Monitors

The construction of runtime monitors for environment assumptions is shown exemplary in this section on the basis of the environment assumptions clause of the extended safety requirement ESR1 specified in the Table 5.8. Notice that the environment assumption in ESR1 requires that with a probability of at least 0.75, the VSE function receives a valid GPS signal with an error of less an 0.15 (cf. Table 5.4). This means that, according to this environment assumption, at least 75% of the GPS data received by the ego vehicle's GPS sensors is represented by valid GPS signals, which carry a data error of at most 0.15 $\frac{m}{s}$ for the GPS-measured vehicle speed. Equation (5.37) formalizes this environment assumption in PCTL.

$$P_{\geq 0.75}(G\ [(s_{GPS} == \mathit{true}) \land (e_{GPS} \leq 0.15)]) \tag{5.37}$$

The environment assumption in Equation (5.37) can be rewritten as in Equation (5.38):

$$P_{\geq 0.75}(G\ \psi_{GPSError}) \tag{5.38}$$

where $\psi_{GPSError}$ denotes the event in which the ego vehicle's GPS sensors receive a valid GPS signal with a GPS data error of at most 0.15 $\frac{m}{s}$:

$$\psi_{GPSError} \overset{\text{def}}{=} (s_{GPS} == \mathit{true}) \land (e_{GPS} \leq 0.15) \tag{5.39}$$

Notice that probabilities are expressed over theoretically an infinite number of execution steps. However, the execution of the environment assumptions monitor at runtime is finite. Therefore, instead of probabilities of occurrence of a given event, the relative frequency of occurrence of the respective event is computed (cf. Section 4.5.5).

The environment assumption monitor is defined in Equation (5.40) according to the Definition 4.5.1 (cf. Section 4.5.5):

$$M_{\psi_{GPSError}} : G\ (f^{GPSError}_{Observed} \geq p^{GPSError}_{Assumed}) \tag{5.40}$$

where $f^{GPSError}_{Observed}$ represents the relative frequency of occurrence and $p^{GPSError}_{Assumed}$ is the assumed probability of occurrence for the event $\psi_{GPSError}$. The assumed probability of occurrence for the event $\psi_{GPSError}$ is specified in ESR1 as $p^{GPSError}_{Assumed} = 0.75$, while the relative frequency of occurrence $f^{GPSError}_{Observed}$ is computed as shown in Equation (5.41):

$$f^{GPSError}_{Observed} = \frac{n^{GPSError}_{Observed}}{N} \tag{5.41}$$

where N represents the total number of trials or events taken into consideration and $n^{GPSError}_{Observed}$ represent the absolute frequency of occurrence for the event $\psi_{GPSError}$.

5.2.5. System Implementation

The VSE function is implemented in Matlab/Simulink. The physical environment of the function is the outdoors physical environment in which the vehicle drives. Two inputs from the physical environment measured by the respective vehicle sensors are relevant for the speed estimation function: the road inclination and the GPS sensor data which comprises the GPS-measured vehicle speed and the GPS error associated with it. The technical environment of the speed estimation function is represented by all the vehicle hardware and software components which communicate through their interfaces with the speed estimation function. In this case, it is the ESC component which provides the measurements of the vehicle wheel speed as well as the longitudinal and lateral acceleration of the vehicle. The inputs from the physical environment as well as the ones from the technical environment are sent via a CAN bus to the speed estimation function. A specific component, also implemented in Matlab/Simulink, maps the input data received through the CAN bus from the technical and physical environment of the speed estimation function. This component has been provided in the course of the research project by the automotive OEM, who acted as partner in the project.

5.2.6. System Test

During system test, the goals are to test the speed estimation function itself as well as the environment assumptions monitors. The test of the speed estimation function is done with the help of the test oracle defined for the its safety requirement. Through the test of the safety requirement monitor it can be checked whether the speed estimation function satisfies the safety requirement specification in the controlled environment. The test of the environment assumptions monitors checks whether the monitors are able to detect violations of the specific environment assumptions. Since the system test takes place in a controlled environment, the test engineers can devise specific tests in order to trigger the environment assumptions monitors, e.g., relative frequency of occurrence of a large GPS error, i.e., $e_{GPS} \geq 0.15$, is larger than 0.25 for a finite number of execution steps of the VSE function. System test cases can be defined manually based on expert domain knowledge or can be built based on counterexamples automatically generated through model checking the abstract model of the VSE function. Chapter 2 discusses several methods for the automated generation of test cases for probabilistic and non-probabilistic systems. Specific test drives with test vehicles can be used in order to gather input data for the developed test cases.

For the system test of the speed estimation function, the test engineers at the automotive OEM partner collected data using the OEM's own field test vehicles. The collected test data corresponds to two driving scenarios: (1) smooth driving and (2) dynamic, sportive driving. The traveling time corresponding to the collected data amounts to 1000 s, which represents approximately 16.7 minutes.

Scenario 1: Smooth Driving

The first scenario illustrates the ideal situation, in which the driver adopts a smooth driving style, with no abrupt acceleration or braking. Figure 5.7 depicts the monitor traces of the environment assumptions monitor regarding the GPS data error.

With respect to the monitor for the GPS data error, there are several data points in which a large GPS error, situated above the maximum assumed GPS error, is visible. However, the relative frequency of the data points showing small GPS error is larger than the assumed probability of occurrence, which is why the environment assumption for the GPS data error is still valid.

The first scenario is depicted in Figure 5.8 through two graphics. The upper graphic shows (1) the 2D GPS-measured vehicle speed, (2) the ADMA-measured vehicle speed, which is considered to be the ground truth, and (3) the upper bound for the vehicle speed allowed by the Euro NCAP requirement. The lower graphic in Figure 5.8 shows the test oracle for the safety requirement of the VSE function. The test oracle compares the deviation between the displayed speed and the actual vehicle speed with the maximum speed deviation permitted by the Euro NCAP requirement.

Notice that the scenario illustrated in Figure 5.8 falls right in the margins specified by the Euro NCAP requirement and preconditions. The value range of the actual vehicle speed is situated between 50 $\frac{km}{h}$ and 130 $\frac{km}{h}$. The ADMA speed curve depicts a relatively smooth driving style, with clear-cut acceleration at about 410 s into the trip and deceleration segments at roughly 580 s and 640 s into the trip and continuous periods of time of driving with almost constant speed. The performance evaluation of the VSE function shows that in the first scenario the vehicle speed deviation $v_{Display} - v_{Real}$ is

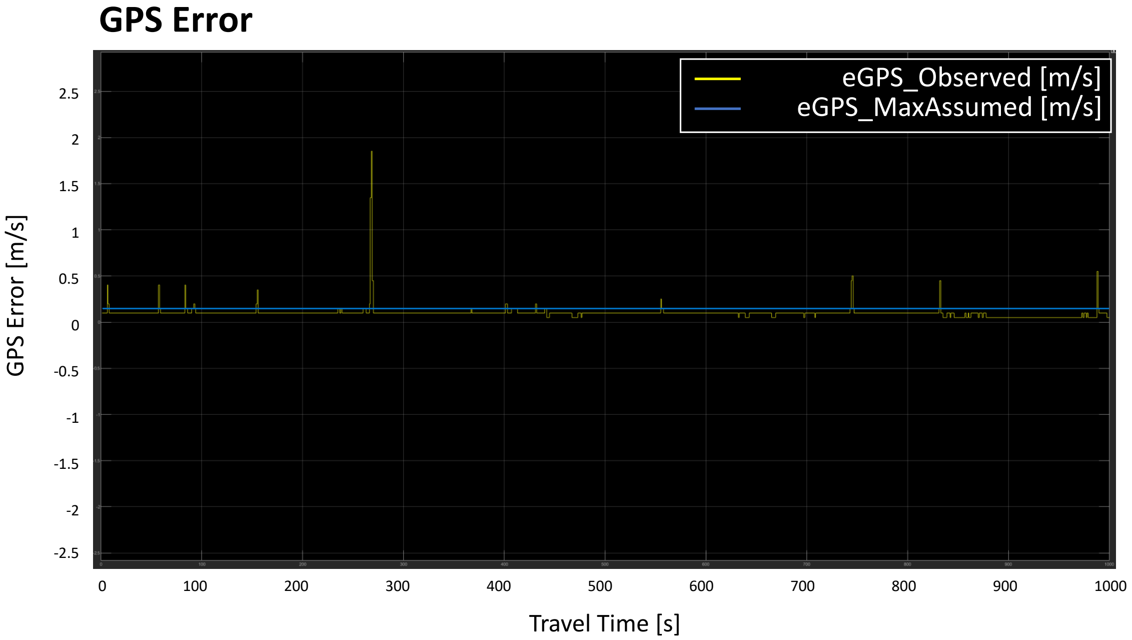


Figure 5.7.: Vehicle Speed Estimation Function: Environment Assumption Monitor for the GPS Data Error during the *Smooth Driving* Scenario.

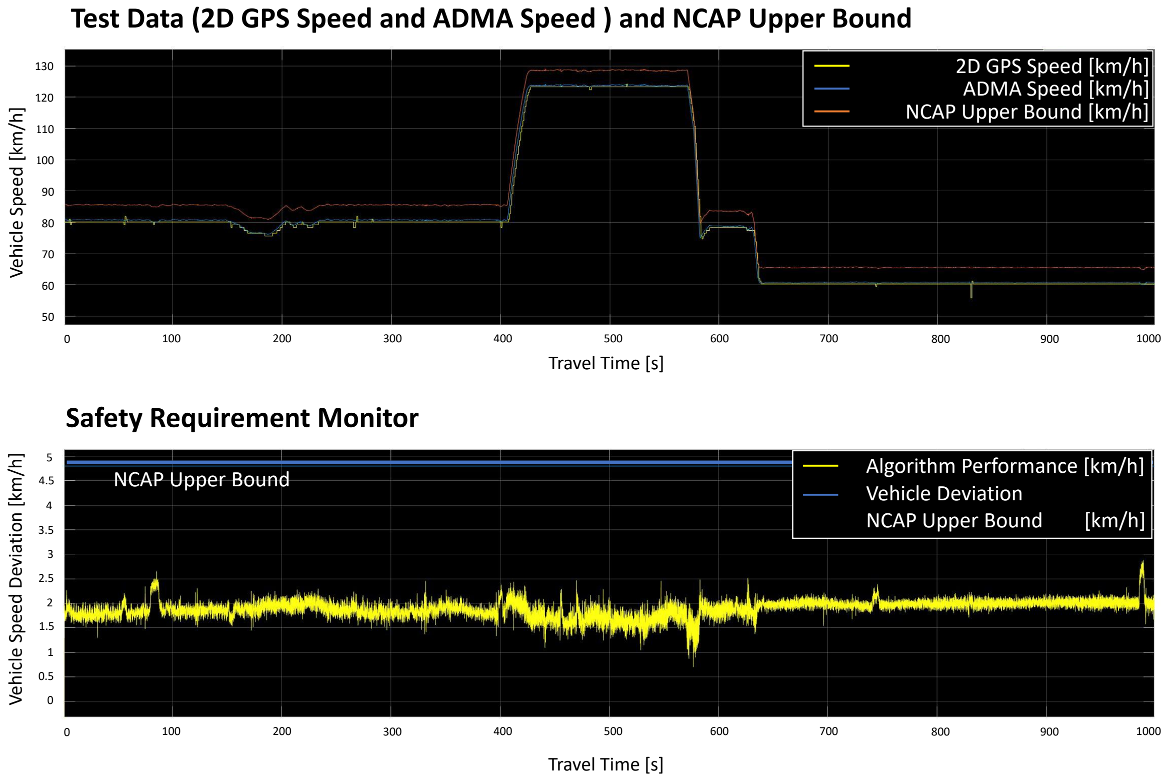
between ca. 0.5 $\frac{km}{h}$ and ca. 3.0 $\frac{km}{h}$, which satisfies the Euro NCAP requirement specified in Equation (5.7).

Scenario 2: Dynamic Driving

The second scenario depicts a situation, in which the driver adopts a more dynamic driving style, with abrupt acceleration and sudden brakes, which alternate frequently throughout the first 250 s of the trip. Figure 5.9 illustrates the monitor traces for the environment assumption monitor regarding the GPS data error.

Similarly to the smooth driving scenario, there are several data points in which a large GPS error, situated above the maximum assumed GPS error, is visible. However, the relative frequency of the data points showing small GPS error is significantly larger than the assumed probability of occurrence, which is why the environment assumption for the GPS data error is still valid.

Similarly to the first scenario, the second scenario is depicted in Figure 5.10 through two graphics. The upper graphic shows (1) the 2D GPS speed, (2) the ADMA speed, and (3) upper bound for the vehicle speed allowed by the Euro NCAP requirement. The lower graphic in Figure 5.10 shows the trace of the safety requirement monitor which depicts the vehicle speed deviation as estimated by the speed estimation function in relation of the maximum speed deviation between the displayed speed and the actual vehicle speed permitted by the Euro NCAP requirement.



Test Data (2D GPS Speed and ADMA Speed) and NCAP Upper Bound

Safety Requirement Monitor

Figure 5.8.: Vehicle Speed Estimation Function: Test Data and Test Oracle of the Safety Requirement in the *Smooth Driving* Scenario (cf. [AZR20]).

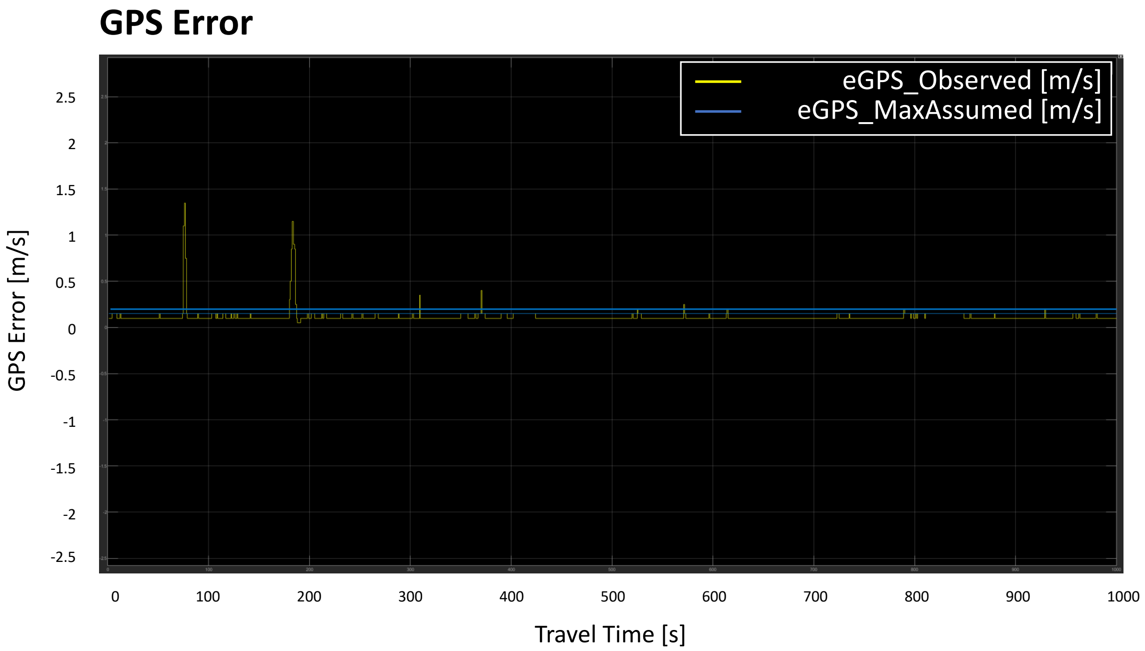


Figure 5.9.: Vehicle Speed Estimation Function: Environment Assumption Monitor for the GPS Data Error during the *Dynamic Driving* Scenario.

The abrupt changes in the curve of the vehicle's longitudinal acceleration are reflected also in the curve of the vehicle speed measured through the vehicle's standard GPS sensors and through the ADMA sensor. The ADMA speed curve has a value range between 0 $\frac{km}{h}$ and 180 $\frac{km}{h}$ throughout the vehicle's trip. In the first 230 s, the ADMA speed oscillates in the interval $[0, 120]$ $\frac{km}{h}$, with abrupt speed-ups and sharp brakes, which alternate frequently. Notice that, after approximately 60 s, the estimation of vehicle speed deviation $v_{Display} - v_{Real}$ is stabilized between a minimum of ca. 0.5 $\frac{km}{h}$ and a maximum of ca. 3.5 $\frac{km}{h}$, thus satisfying the Euro NCAP requirement.

Notice that, during the first 100 s at about 50 s into the vehicle's trip, the estimated vehicle speed deviation drops down to -1 $\frac{km}{h}$, meaning that the actual vehicle speed is underestimated. This occurrence can be attributed to the fact that, at that time, the ADMA speed decreases down to 0 $\frac{km}{h}$, i.e., the vehicle has stopped. No valid wheel speed measurements can be collected while the vehicle is stopped. In this case, the test oracle of the safety requirement reports a violation of the Euro NCAP requirement.

5.2.7. Requirements Validation

Requirements validation is the last stage of testing before the system under test becomes operational. In this phase, acceptance tests are carried out together with the customer in order to see whether the acceptance criteria defined in advance are met by the system under test. Such tests are carried out both in the physical test environment of the system developer as well as in the operational environment at customer's site (cf. Section 3.8). At

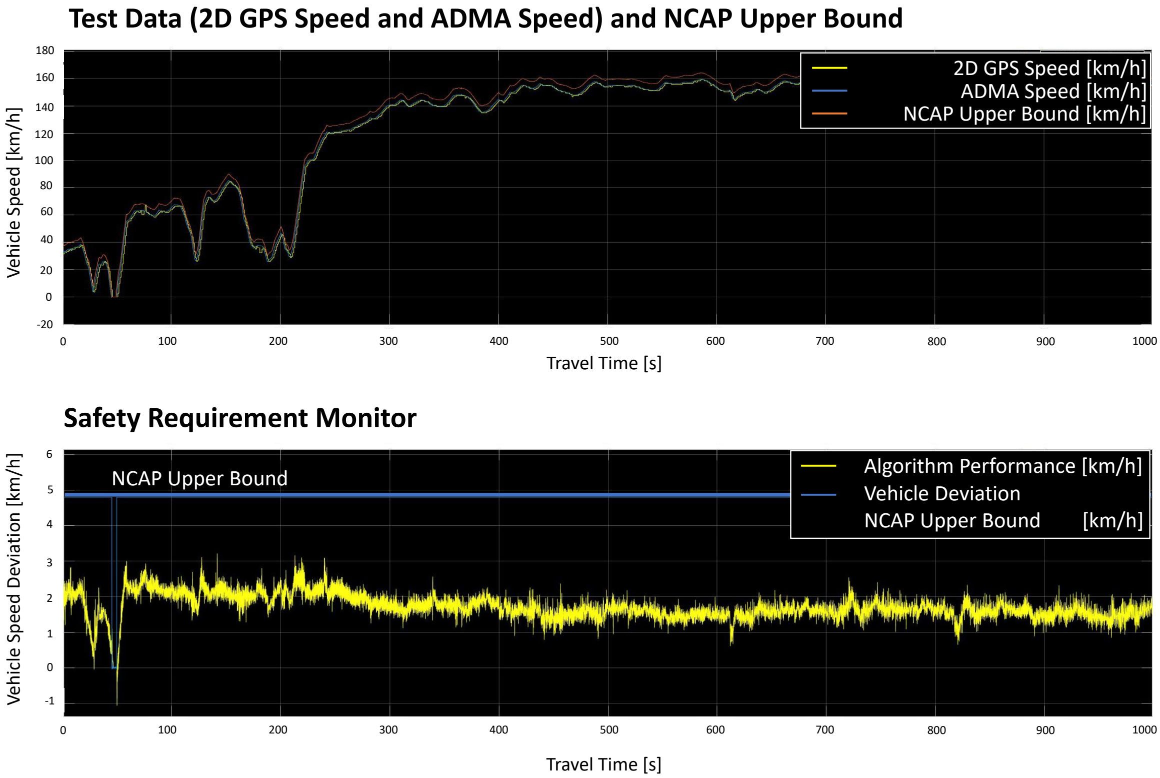


Figure 5.10.: Vehicle Speed Estimation Function: Test Data and Test Oracle of the Safety Requirement in the *Dynamic Driving* Scenario (cf. [AZR20]).

the end of the project *Accurate Vehicle Speed Estimation*, the speed estimation function was still under development, awaiting a preliminary positive review from the certification authority so that it could enter the series production phase, in which the function is integrated on the target hardware and extensive hardware and software integration tests are carried out. Therefore, no requirements validation took place during the timeline of the project.

5.3. Summary

The goal of this chapter was to evaluate the feasibility of the RMEA concept on real-world systems. Thus, this chapter presented two case studies built around two such systems. The first case study is built around a mobile service robot commissioned to autonomously execute transportation tasks in a hospital environment. The second case study investigates the feasibility of the RMEA concept on an automotive system function in charge of estimating and displaying the speed of a moving vehicle on the vehicle's instrument board.

The applicability and feasibility of the RMEA concept for real-world safety-critical systems were demonstrated through the two case studies presented in this chapter. For each case study, the application of the RMEA concept discussed in detail the artifacts that are produced during the different phases of the system development process:

1. Catalog of functional and safety requirements,
2. Catalog of safety hazards derived through HARA analysis,
3. Catalog of environment assumptions that cover the respective safety hazards,
4. Design-time models of the system under analysis and its environment
5. Runtime monitors for the environment assumptions, and
6. Test cases used to test both the system under analysis and the environment assumptions monitors.

Chapter 6.

Contributions with respect to Related Work

6.1. Obtaining Environment Assumptions 233

6.1.1. Manual Methods for Specification of Environment Assumptions . 234

6.1.2. Automated Methods for Obtaining Environment Assumptions . . 237

6.2. Assumptions in Verification and Validation Processes 241

6.2.1. Assumptions in Design-time Verification 241

6.2.2. Controller Synthesis and Environment Assumptions 243

6.2.3. Combining Design-time Verification with Runtime Validation . . . 245

6.3. Comparison of the Proposed Approach with Related Work 249

6.3.1. Comparison with the Manual Methods for Environment Assumptions Specification 249

6.3.2. Comparison with Automated Methods for Obtaining Environment Assumptions 251

6.3.3. Comparison with Methods that integrate Assumptions in Verification and Validation Processes 252

6.4. Summary 256

The goal of this chapter is to analyze the related state-of-the-art research work in the scope of this thesis, compare it with the approach presented in this thesis and then emphasize the contributions brought forth by this thesis in relation of the related research. Section 6.1 presents various approaches by which environment assumptions can be obtained, while Section 6.2 discusses approaches for the verification and validation of environment assumptions. In Section 6.3, the approach used in this thesis is compared with the techniques found in related research. Section 6.4 concludes this chapter with a short summary of its findings.

6.1. Obtaining Environment Assumptions

Environment assumptions have to be explicitly defined in order to carry out design-time verification and runtime validation of automated safety-critical systems. Environment assumptions can be obtained in two distinct ways: (1) either they are directly specified by experts with specific domain knowledge (cf. Section 6.1.1), or (2) they are learned or synthesized from monitor observations of the system's environment (cf. Section 6.1.2).

6.1.1. Manual Methods for Specification of Environment Assumptions

The concept of assumption, formulated with respect to the environment of a system or a system's component, is rooted in compositional verification (cf. [AL93]). Two research areas in which assumptions play an important role have distinguished themselves throughout the years: assume-guarantee contracts and agent-based systems.

Assume-Guarantee Contracts

Assume-guarantee contracts are technique in the spectrum of compositional verification, more specifically of A/G reasoning (cf. Section 2.5.10). Benveniste et al. [BCP07] propose a formal concept for A/G contracts with application for safety-critical embedded systems. Their concept allows the description of component interfaces and component behavior, as well as the composition of components. A component consists of an interface, an expected behavior, and one or more implementations of this behavior (cf. [BCP07]). A/G contracts are used to describe the expected component behavior using pairs of assumptions and guarantees defined over the set of ports and flows specified in the component interface. The concept of A/G contracts in [BCP07] is extended with probabilities in [DC08]. A probabilistic contract has associated with it a probability distribution, that is defined over the history of values received on the component's ports that are not controlled by the contract. The system models targeted in [BCP07] and [DC08] are non-probabilistic. In contrast, [Del10] and [DCL11] extend the probabilistic A/G contracts to be applicable also to probabilistic systems. The system under analysis and the contracts formulated on its components are represented by sets of system executions or are described through formal logics, e.g, LTL or a quantitative variant of CTL introduced by Alfaro in [dAFH$^+$04], called discounted CTL (cf. [Del10], [DCL11]). The system taken under consideration in [Del10] is the precision time protocol (PTP), which is a protocol used for the synchronization of clocks in a computer network.

Arts et al. [ADT14] use A/G contracts for the verification of the communication protection mechanism defined in the AUTOSAR standard [1]. This protection mechanism is defined as a library of functions that detect communication faults, e.g., message loss or message corruption, in the transmission of safety-critical data between the ECUs of a vehicle (cf. [ADT14]). The system under analysis which uses the communication protection mechanism of the AUTOSAR standard is a simplified version of an airbag system (cf. [ADT14]). The system components are modeled in the Smv language and the A/G contracts defined for each component are specified in LTL with past operators. The behavior of each component is verified against its A/G contract using the NuSmv model

[1] https://www.autosar.org

checker. The state machine describing the components' behavior is used to generate test cases in order to ensure compliance with its implementation in the C language (cf. [ADT14]).

Li [LNSV$^+$17] and Nuzzo [NLSV$^+$19] use A/G contracts to verify closed-loop control systems with probabilistic requirements. The probabilistic constraints used in the assumption and guarantee of an A/G contract are specified with stochastic signal temporal logic (StSTL). Verification tasks of bounded StSTL contracts, e.g., compatibility, consistency, or refinement checking, as well as synthesis of model predictive control strategies are reformulated as mixed integer optimization problems. The approach is applied on an aircraft power distribution system.

In [Gle14], the A/G safety goals of a technical system are obtained through safety analysis. The safety goals are behavioral properties that are defined for each hazard identified through hazard analysis. The technical system and its environment are modeled as two agents. Each agent can control their own state and actions and can monitor the actions and state of the other agent. Formally, the agents are encoded as mode transition systems and the behavioral properties are specified with PCTL* (cf. [Gle14]). Each A/G pair of a safety goal has its guarantee assigned to the functions of the system agent and the assumption to the tactics of the environment agent. Based on the specified safety goals, there are safety measures defined that contain the corresponding safety modes and actions (cf. [Gle14]).

Assume-guarantee pairs are used also in [GFN19] to verify models of cyber-physical machines (CPM). The verification results are delivered within formal model-based assurance cases (FMACs). An FMAC is an assurance case which uses a formal model to derive evidence for the top-level claims. The FMAC consists of the specification of a CPM, which describes the system behavior, a set of assumptions and a set of guarantees (cf. [GFN19]). A system is formalized as a generalized hybrid program, that can be specified with differential dynamic logic (dL) [Pla08]. The assumptions are defined as predicates over the monitored variables by the CPM, while the guarantees are defined as predicates over the internal variables of the CPM as well as the variables which are monitored and controlled by the CPM (cf. [GFN19]). The approach defines two patterns for assurance cases. One pattern is for the construction of assurance cases and the other pattern is for their extension. The extension of an assurance case happens through the refinement of the CPM model, the weakening of its assumptions and the addition of further guarantees. Both patterns show how assurance results from previous engineering steps can be preserved (cf. [GFN19]). The approach is applied to a mobile ground robot. The mobile ground robot is modeled in a formal modeling language which implements the differential dynamic logic (dL) of [Pla08] in ISABELLE/UTP [FBC$^+$20]. The patterns are formalized in ISABELLE/SACM [NFGK], while the proofs are carried out with ISABELLE/HOL [Nip02].

Assumptions in Agent-based Systems

Verification of agent-based systems is another area in which the concept of assumption is important. Rational agents represent a computation model which is used to describe the functioning of decision-making components in hybrid autonomous systems (cf. [FDW13]). The rational agent makes discrete decisions, which are then translated into continuous control actions of the hybrid autonomous system. In order to verify such components, the respective agents are modeled in the language Gwendolen [DM08], while the properties to be verified are specified in a belief-desires-intentions (BDI) logic [Rao96]. The BDI logic is in fact LTL, extended with specific modal operators, that are used to check the agent's beliefs, goals, actions and intentions (cf. [FDW13]). The verification is carried out with the model checker Agent Java Path Finder (AJPF) [DFWB12], which extends the model checker Java Path Finder (JPF) 2 with mechanism for the checking of agent properties. The AJPF model checker together with the agent programming language Gwendolen and the BDI logic are subsumed to the MCAPL framework, which is used for the verification of agent-based systems modeled in Gwendolen (cf. [FDW13]). This framework is applied in several use case scenarios in urban search and rescue, autonomous satellites and adaptive cruise control systems (cf. [DFL$^+$16]). In each use case scenario, environment assumptions are formulated as logical statements in the BDI logic in order to derive further properties of the system under analysis. In the scenario of urban search and rescue, the environment assumptions refer to the robot sensors and actuators, e.g., whether the its sensors detect a human correctly and whether its motor control functions correctly (cf. [DFL$^+$16]). In the autonomous satellite scenario, assumptions are formulated with respect to different components, e.g., the planing component, which is assumed to deliver a plan to get the satellite in a desired position on the agent's request (cf. [DFL$^+$16]). The scenario of the adaptive cruise control, inspired by the one presented in [LPN11], makes assumptions with respect to the distance between the ego vehicle and the leading vehicle and with regard to the presence of another vehicle on the left lane before the ego vehicle changing to that lane (cf. [DFL$^+$16]). Instead of being formalized explicitly as logical statements in an extension of LTL, the assumptions are sometimes considered as part of the environment model (cf. [FDA$^+$18b]). The environment model is generated from trace expressions formalized in Prolog and is used for verification of the agent with the model checker AJPF (cf. [FDA$^+$18b]). The trace expressions used to generate the environment model are also used to generate runtime monitors, that are used to check whether the real environment validates the assumptions made in the environment model (cf. [FDA$^+$18b]).

Rational agents are also used to model a component in charge of high-level control and autonomous decision-making in the Curiosity Mars rover (cf. [CFL$^+$20]). The agent-environment interface describes the decisions of the agent in response to inputs received from the environment. This interface is modeled in Dafny [Lei10], using preconditions and postconditions as well as loop invariants (cf. [CFL$^+$20]). The rational agent is programmed in Gwendolen, while the other components in the Mars rover that interact with the agent are implemented in Ros. The communication between the rational agent and the Ros components is implemented via a client-server model similar to Ros services (cf. [CFL$^+$20]). Different tools are used to verify the rational agent, its interface with its environment and underlying communication between the client and server nodes. Thus, the rational agent is verified with the AJPF model checker against properties expressed in BDI logic. The agent-environment interface modeled in Dafny defines specific functions that can access the information received from the environment, e.g., wind speed or radiation. It enables the verification of safety properties regarding the agent's behavior under the premise that the information received from the environment is interpreted correctly by the agent (cf. [CFL$^+$20]). The verification of the agent-environment interface is carried out with the automated theorem prover Z3 [dMB08], that is integrated in the Dafny tool. The assumptions used in the Dafny model are checked at runtime using the framework ROSMonitoring [FCF$^+$21]. The monitors are synthesized from properties written in Runtime Monitoring Language (RML) [Fra20].

A further approach which uses rational agents to model autonomous control systems is Limited Instruction Set Agent (LISA), proposed in [IQV16]. LISA allows the definition of predicates for sensory perception and action feedback, which allow the agent to be aware of the consequences of its own actions (cf. [IQV16]). Along with with its sensory perception and action feedback, the agent's goals are considered to be part of its set of beliefs (cf. [IQV16]). A rational agent modeled in LISA can be compiled into a DTMC or an MDP. A transition in a DTMC corresponds to an action of the agent's plan triggered by a change in the agent's beliefs, in which changes of action feedback and sensory perception are given with known probability distributions (cf. [IQV16]). A transition in an MDP is similar to a transition in a DTMC, except that the agent's plan selection involves a nondeterministic choice, because the number of applicable plans is larger than the number of triggering events (cf. [IQV16]). The authors propose to use the generated Prism model both for design-time and runtime verification (cf. [IQV16]). Their proposal is to carry out the runtime verification through the agent's function for plan selection. In this sense, the plan selection function of the rational agent is considered to take as inputs the current set of beliefs and desires of the agent and produce as output one plan that is most likely to succeed based on the set of the agent's desires (cf. [IQV16]). For the selection of the agent's plan, the authors of [IQV16] propose to use the probability values resulted from the verification of the Prism model against predefined queries (cf. [IQV16]). The LISA approach is shown on the example of an autonomous surface vehicle used for mine detection and disposal (cf. [IQV16]).

6.1.2. Automated Methods for Obtaining Environment Assumptions

The research discussed in Section 6.1.1 represents a range of approaches in which the environment assumptions are specified manually. Defining relevant environment assumptions requires a high level of knowledge, often at expert level, about the system under analysis and its environment. This often means a white box view of the system, in which the system designer has a thorough understanding of the system behavior in the system's environment. There are, however, other ways of obtaining environment assumptions. One approach is to learn the assumptions under which the system under analysis satisfies its system requirements. Another way is through the observation of the system behavior in its environment. This section presents learning-based approaches for the generation of assumptions and specification mining approaches, that are used to extract the relevant environment assumptions from the observation of the system behavior in its environment.

Learning-based Methods for Assumptions Generation

Learning-based assumptions generation has been introduced for non-probabilistic systems and discussed extensively in [CGP03], [GPB05], [PG06], [GGP07], and [PGGB$^+$08]. It is a compositional verification technique, in the spectrum of assume-guarantee reasoning. This technique relies on the learning algorithm L* and model checking, in order to learn assumptions in an iterative manner. A brief intuition on the workings of this technique is given on the example of an asymmetric proof rule for non-probabilistic systems with two components in Section 2.5.2. The approach is complete, because it ensures that if the system satisfies its property, then there always exists an assumption that can be used to carry out the verification in a compositional manner (cf. [PGGB$^+$08]). This approach is demonstrated on several subsystems developed at NASA for the control of planetary rovers (cf. [CGP03], [PG06]). In the application of this approach, the systems are represented as finite labeled transition systems. The learned assumptions are deterministic finite state machines encoded as such in the Ltsa tool (cf. [CGP03]) or as processes encoded in the modeling language Promela of the Spin model checker (cf. [PG06]).

An alternative approach to learning-based assumption generation is assume-guarantee abstraction refinement (AGAR) (cf. [GBPG08]). For a system $M_1||M_2$, the AGAR approach computes assumptions as conservative abstractions of the interface behavior of M_1. To give an intuition of how AGAR works, consider the proof rule Asymm in Section 2.5.2. In each iteration, the computed assumption A satisfies ψ_1 by construction and it is only verified against ψ_2 (cf. [GBPG08]). If the verification succeeds, then it is shown that the system $M_1 \mid\mid M_2$ satisfies the property G. Otherwise, the generated counterexample is analyzed to see if it is caused by over-approximation in the abstraction or if it is related to an actual error in the system $M_1 \mid\mid M_2$. In case of a spurious counterexample, this is used to refine the assumption A (cf. [GBPG08]). This approach is similar to counterexample-guided abstraction refinement (CEGAR) [CGJ$^+$00] with a few differences between the two approaches. Counterexample-guided abstraction refinement computes abstractions of programs based on a set of abstraction predicates, using spurious counterexamples to refine these abstractions. The abstractions computed with CEGAR are over-approximations of the program under analysis (cf. [CGJ$^+$00]). By comparison, AGAR works on actions of finite automata and computes under-approximations of these finite automata, which represent the assumptions used for A/G reasoning (cf. [PGGB$^+$08]). In contrast to the learning-based assumption generation method in [CGP03] and [PGGB$^+$08], AGAR does not require that the computed assumptions are deterministic, which may result in smaller assumptions (cf. [GBPG08]). Smaller assumptions are further obtained by combining AGAR with alphabet refinement (cf. [GBPG08] and [PGGB$^+$08]).

Feng et al. [FKP10] adapt the approach in [PGGB$^+$08] for learning assumptions for asynchronous probabilistic systems and demonstrate this approach for the asymmetric proof rule. Given a probabilistic safety property $\langle G\rangle_{\geq p_G}$ and a system $M_1 \mid\mid M_2$ consisting of the parallel composition of two components M_1 and M_2, the approach attempts to find the probabilistic assumption $\langle A\rangle_{\geq p_A}$ that can be used to verify $M_1 \mid\mid M_2$ with respect to the property $\langle G\rangle_{\geq p_G}$ (cf. [FKP10]). Both components M_1 and M_2 are modeled as MDPs.

The assumption $\langle A \rangle_{\geq p_A}$ and the property $\langle G \rangle_{\geq p_G}$ are probabilistic safety properties, which are represented as deterministic finite automata (cf. [FKP10]). Similar to the work in [PGGB$^+$08], the algorithm L* interacts with a teacher through membership queries and conjectures, in order to establish whether a word is an element of the target language, and respectively, whether the deterministic finite automata hypothesized by the L* algorithm accepts the target language (cf. [FKP10]). The deterministic finite automata conjectured by the L* algorithm represents an assumption, which is later used to verify the second premise of the asymmetric proof rule, $\langle A \rangle_{\geq p_A} M_2 \langle G \rangle_{\geq p_G}$, with the probabilistic model checker Prism. If the triple $\langle A \rangle_{\geq p_A} M_2 \langle G \rangle_{\geq p_G}$ is disproved, then teacher provides a counterexample, which is further used to refine the assumption (cf. [FKP10]). A triple $\langle A \rangle_{\geq p_A} M \langle G \rangle_{\geq p_G}$ is disproved if there exist a scheduler σ of $M[Act_A]$ that satisfies the assumption $\langle A \rangle_{\geq p_A}$ but disproves the guarantee $\langle G \rangle_{\geq p_G}$ (cf. Section 2.5.2). The counterexample generated by the teacher has the form (σ, w, c), where σ is a finite-memory scheduler, (σ, w) is a witness for $\langle A \rangle_{\geq p_A}$ and (σ, c) is a counterexample for $\langle G \rangle_{\geq p_G}$ (cf. [FKP10]). The presented approach is quantitative, as it computes lower and upper bounds on the probability p_G of G being satisfied (cf. [FKP10]). The approach is applied to the case study of a resource arbiter module adapted from [PG06], that is part of the control software of planetary rovers (cf. [Fen14]). In [FHKP11], this approach is adapted to learn probabilistic assumptions for synchronous probabilistic systems. The assumptions are represented as probabilistic finite automata, while the components of the system are represented as probabilistic input/output automata, whose parallel composition results in a DTMC (cf. [FHKP11]). In addition, the work in [Fen14] presents a method for learning assumptions for probabilistic systems, whose components are DTMCs encoded as boolean formulae. In turn, the assumptions are represented as interval DTMCs (cf. [Fen14]). Notice that, although learning assumptions for non-probabilistic systems is complete (cf. [PGGB$^+$08]), the learning approach applied for probabilistic systems may not yield any probabilistic assumption, as this assumption may not exist.

Komuravelli et al. [KPC12] apply the AGAR approach to compute assumptions for labeled probabilistic transition systems (LPTS). Such systems display both probabilistic and nondeterministic behavior, thus essentially being equivalent to a PA (cf. [KNPQ13]). In contrast to non-probabilistic systems in which the counterexamples are traces in finite labeled transition systems (cf. [GBPG08]), in an LPTS counterexamples are stochastic trees obtained during failed simulation checks. These counterexamples are further used to refine the inferred assumptions. Both the generated assumptions as well as the system property to be verified are represented as LPTS (cf. [KPC12]).

Specification Mining Methods for Assumptions Generation

Besides learning-based methods and AGAR, specification mining techniques represent another possibility for the automated generation of assumptions. Specification mining techniques can be used to generate additional assumptions, that are added to an original property specification in order to make it realizable. A formal property specification is realizable if a correct-by-construction controller can be generated through synthesis from it, otherwise the respective property specification is not realizable. An under-constrained environment or an over-constrained system can lead to an unrealizable property specification (cf. [Li14]). An under-constrained environment means that the assumptions expressed about the environment are too weak. An over-constrained system means that the guarantees which the system has to hold are too strong. Li proposes a specification mining technique which relies on synthesis from General Reactivity (GR(1)) formulae (cf. [LDS11], [Li14]). GR(1) is a strict A/G fragment of LTL, i.e., GR(1) formulae have the general form $\psi_e \rightarrow \psi_s$, where ψ_e represents the environment assumption, and ψ_s is the system guarantee (cf. [MR15]). A controller synthesized from a given specification fulfills the respective specification by construction. Thus, the guarantee ψ_s holds only when ψ_e is valid (cf. [WTM12]). The specification mining method proposed by Li [Li14] takes a library of GR(1) specification templates and a counter-strategy as input and produces a candidate assumption as output. In the first step of this approach, GR(1) synthesis is carried out on the original property specification. If it is not realizable, then the GR(1) synthesis produces a counter-strategy, that contains the next steps executed by the environment agent in response to the current output of the system agent (cf. [Li14]). The second step of this approach uses the library of GR(1) specification templates and applies specification mining in an attempt to find a specification in the library that is satisfied by the counter-strategy of the environment agent (cf. [Li14]). In the third step, the property specification mined out of the GR(1) templates library is negated and added to the original property specification in order to make the latter realizable (cf. [Li14]). Notice that this works because of the law of double negation, i.e., since the mined property specification is satisfied by the counter-strategy of the environment agent, adding its negation to the original property specification constrains the environment agent in such a way that the new property specification may be realizable, i.e., an implementation for a system controller can be generated from it. The process consisting of the three steps discussed above is carried out in an iterative manner, until all specifications in the library of GR(1) specification templates have been checked out or the resulting specification is realizable (cf. [Li14]).

In [LSSS14], Li uses the same approach to synthesize strategies for a human-in-the-loop controller from a GR(1) fragment of LTL for SAE level 3 vehicle automation. The synthesized human-in-the-loop controller consists of an automatic controller, a human operator and an advisory control mechanism that orchestrates the sharing of responsibilities between the automated controller and the human operator (cf. [LSSS14]).

Introspective environment modeling is an approach introduced by Seshia in order to deal with the uncertainties related to the dynamics of the environment model and those regarding the state of environment objects (cf. [SS16], [Ses19]). In this approach, the system's behavior and its interface to its environment are analyzed in order to extract a representation of the environment in which the correct functioning of the system with respect to a formal property specification is guaranteed (cf. [Ses19]). The problem of introspective environment modeling is reduced to the problem of synthesizing a controller from a GR(1) specification. It uses the approach in [LDS11], in which counter-strategies of the environment are automatically generated through reactive synthesis applied on an unrealizable property specification. The generated counter-strategy constitutes the basis of the environment assumption, which makes the resulting property specification realizable. The approach is demonstrated on a simplified autonomous driving scenario (cf. [Ses19]).

In another research work, Krismayer [Kri20] proposes an approach for mining system constraints for complex systems, e.g., systems of systems, on the basis of system execution logs. Since the proposed approach allows for different input formats, the system execution logs are parsed in order to obtain a uniform event format, used for the subsequent steps of the constraint mining process (cf. [Kri20]). From the resulting event logs, the proposed mining method identifies various types of event sequences which allows mining of temporal constraints with respect to event occurrence, event order, and event timing (cf. [Kri20]). In the next step of the constraint mining approach, feature vectors containing event data are extracted from each event sequence. The feature vectors serve as a basis to derive value constraints as well as hybrid constraints, the latter type of constraints being a combination between temporal and value constraints (cf. [Kri20]). The resulting constraints are filtered, grouped and ranked, according to predefined criteria, e.g., duplicate constraints are removed, similar constraints are grouped together, and constraints are ranked according to their confidence (cf. [Kri20]).

6.2. Assumptions in Verification and Validation Processes

While Section 6.1 focused on reviewing methods for obtaining environment assumptions, this section takes a look at related work that illustrate the usage of assumptions in verification and validation processes. This section reviews related work in which environment assumptions are used in relation with design-time verification approaches (cf. Section 6.2.1), as well as research work that address the combination of design-time verification and runtime validation approaches and the role of environment assumptions in the development process (cf. Section 6.2.3). Additionally, related work that bring controller synthesis approaches in relation with environment assumptions is discussed (cf. Section 6.2.2).

6.2.1. Assumptions in Design-time Verification

This section takes a look at design-time verification approaches that are used in relation with assumptions. One of these techniques is theorem proving. Mitsch et al. [SKA13] present the verification of an autonomous robot via the theorem prover KeyMaera with respect to two types of safety properties: (1) passive safety, i.e., the robot does not actively collide with obstacles in its environment, and (2) passive friendly safety, i.e., the robot allows dynamic obstacles in its environment sufficient maneuvering space, so that the obstacles can avoid a collision with it (cf. [SKA13]). The dynamic obstacles in the robot's environment are considered to have an arbitrary continuous motion, with a known upper bound on their velocity (cf. [?]). The robot and the obstacles in its environment are modeled as a hybrid system, which describes both discrete control choices and the continuous physical motion, i.e. discrete computations of actuator values for braking translate into a continuous motion of slowing down. Both the hybrid system and the safety properties are expressed in differential dynamic logic (dL) (cf. [SKA13]).

In [MGVP17], two additional types of safety properties are added for the verification of ground robots: (3) static safety, i.e. the robot does not collide with static obstacles, and (4) passive orientation safety, i.e. imperfect coverage of the robot's sensors is allowed, meaning that not everything in the robot's environment is visible to the robot (cf. [MGVP17]). As in [SKA13], the obstacles are considered to have an arbitrary continuous motion, only with a known upper bound on their velocity. Besides this, there are further specific assumptions made about the obstacle movement, for every type of safety property considered - static safety, passive safety, passive friendly safety, and passive orientation safety (cf. [MGVP17]). Examples of assumptions on the obstacle movement are the known maximum velocity of obstacles in case of passive safety property or that obstacles remain stationary and do not move in case of static safety property (cf. [MGVP17]). Differential dynamic logic is used to model the hybrid system that consists of the robot and the obstacles in its environment as well as to specify the safety properties. The design-time verification is carried out with the theorem prover KeyMaera X. In [CFL$^+$20], the interface between a decision-making component of the Curiosity Mars rover and its environment is modeled as a Dafny model and verified with the Z3 theorem prover, integrated in the Dafny tool.

Besides theorem proving, assumptions are also used in relation with other design-time verification techniques, e.g., model checking. Section 6.1.1 gives an overview of approaches in which model checking is used in obtaining assumptions for automated control systems as well as for the verification of such systems. In some of the presented approaches assumptions are obtained manually for automated control systems modeled with the help of rational agents. In each of these approaches, the verification of such systems is usually carried out with the help of the AJPF model checker (cf. [FDW13], [DFL$^+$16], [FDA$^+$18b]). The Prism model checker is used for the verification of rational agents in which changes of action feedback and sensory perception are given with known probability distributions (cf. [IQV16]). Another way to obtain assumptions is by learning them from observations of the system behavior in its environment (cf. Section 6.1.2). In learning-based assumption generation, a teacher interacts with the learning algorithm L* and provides an answer to the question whether the assumption conjectured by L* is the right assumption or not (cf. Section 6.1.2). The teacher is in fact a model checker which provides the required answer by checking if the system under analysis satisfies its system property under the assumptions conjectured by the L* algorithm (cf. Section 6.1.2). The verification is carried out with different model checkers e.g., LTSA model checker (cf. [CGP03], [GPB05], [GGP07], [GBPG08], and [PGGB$^+$08]) or the Spin model checker (cf. [PG06]). In learning-based assumption generation for probabilistic systems, the verification is carried out with the Prism model checker (cf. [FHKP11], [Fen14]).

Different verification techniques can also be combined to verify modular software systems, e.g., in autonomous space robots (cf. [FCD$^+$19]). The software system of the autonomous robot consists of several subsystems: a vision subsystem, a planner, a plan reasoning agent and a hardware-software interface (cf. [FCD$^+$19]). The subsystems of the robot software system are arranged in a pipeline from the vision subsystem down to the hardware-software interface, and the assumptions of each subsystem are considered to follow from the guarantees of the previous subsystem (cf. [FCD+19]). The A/G properties are formulated in first-order logic (FOL) for each subsystem as high-level node specifications, and LTL to reason about combinations of such specifications (cf. [FCD+19]). Theorem proving is used to verify the planner, while model checking is applied for the verification of the plan reasoning agent (cf. [FCD+19]).

6.2.2. Controller Synthesis and Environment Assumptions

Given a system under analysis, formal verification methods used at design-time, e.g., model checking, verify a formal system model against a formal property specification, and provide an answer as whether the system model is correct with respect to the property specification or not. The formal system model is represented usually as a finite state-transition system, while the property specification is formalized in a temporal logic formula (cf. Section 2.5.2). Rather than verifying an existing system model with respect to a formal property specification, reactive synthesis is an automated procedure to obtain a correct-by-construction system from a temporal logical specification, more explicitly an LTL formula (cf. [MR15]). The system implementation obtained as a result of reactive synthesis is an automaton which accepts inputs from the environment and produces the system output. The assignments of the input and output variables satisfy the property specification on every infinite run of the automaton (cf. [MR15]). The obtained implementation is used to control the system under analysis. Therefore, reactive synthesis is sometimes denoted as controller synthesis (cf. [UPÇ12], [WF12]).

Various synthesis techniques are applied for robot controllers which interact with dynamic environment agents. In [UPÇ12] and [WUB+12], synthesis is used in an incremental manner to compute progressively a set of policies for a robot interacting with several independent environment agents. The dynamic agents in the robot's environment are modeled as Markov chains, while the robot is modeled as a deterministic finite transition system for the deterministic case (cf. [UPÇ12] or as an MDP for the stochastic case (cf. [WUB+12]). The objective for the robot is to maximize the probability of behaving according to a its given specification, which is expressed in a safety fragment of LTL, namely co-safe LTL (cf. [UPÇ12], [WUB+12]). In [WF12], a collection of environment models are available to the robot, with each model corresponding to a different mode of the environment. It is considered that the system is not aware of the current environment mode, that is also subject to change during execution. The robot is modeled as an MDP, while the environment models are depicted as Markov chains. The robot has two objectives, namely to maximize the expected probability and respectively the worst-case probability to satisfy a specification given in LTL. In [WUB+13], the synthesis approach computes control policies for multi-agent systems in which robots interact with each other and with environment agents that may not be completely independent. Both the robots and the environment agents are modeled as MDPs and the specification is given in LTL (cf. [WUB+13]). Chen et al. [CKSW13] take a different approach to controller synthesis. They apply multi-objective model checking in an urban driving scenario, in which the system and the adversarial environment are modeled as the two players of a stochastic two-player game. The multi-objective queries formulated for the stochastic game are conjunctions of LTL formulae or conjunctions of reward functions (cf. [CKSW13]). The approach is implemented with Prism-games [CFK$^+$13], which is an extension of the Prism model checker for stochastic games.

Reactive synthesis underlies two main challenges. On one side, synthesis of controller strategies from LTL specifications has a large worst-case complexity, i.e., double exponential in the length of the formula (cf. [MR15]). On the other side, declarative property specifications which are closer to informal system requirements than LTL formulae are difficult to write using LTL operators (cf. [MR15]). In order to reduce the complexity of controller synthesis for LTL specification, the work in [WTM12] proposes the method of receding horizon, which breaks the synthesis problem into a set of smaller problems. Instead of planing out the robot motion from the initial state to the goal state, the idea of receding horizon is to plan out the execution of an autonomous robot only for a short segment ahead, starting from the currently observed state and without considering all the possible behaviors of the environment (cf. [WTM12]). The specification for the controller synthesis is a GR(1) formula, of the general form $\psi_e \rightarrow \psi_s$, where ψ_e is denoted as the environment assumption, and ψ_s as the system guarantee (cf. [WTM12]). Since the synthesized controller fulfills the specification by construction, the guarantee ψ_s holds only when ψ_e is valid. Should ψ_e be invalidated, the system generates a new goal by removing the problematic transition from the finite transition system (cf. [WTM12]).

To address the issues of reactive synthesis, Maoz et al. [MR15] propose an approach which relies on using the LTL specification patterns identified by Dwyer in [DAC99] and transforming the respective LTL formulae in GR(1) formulae. GR(1) is a fragment of LTL, for which a symbolic synthesis algorithm is provided by Piterman [PPS06] and Bloem [BJP$^+$12]. The goal of the GR(1) synthesis is finding a controller that satisfies a formal property specification given over a set of variables which describe the environment state and respectively the system state (cf. [MR16]). The GR(1) synthesis problem is formulated as a game between a system player and an environment player. The synthesis problem for GR(1) specifications defines the variables which are controlled by the environment player and by the system player, the constraints used to formulate the environment assumption and the system guarantee, as well as the winning condition of the game (cf. [MR15]). Three types of constraints are used for the formulation of the environment assumption and the system guarantee: (1) constraints placed on the initial states of the automaton, (2) safety goals, and (3) liveness or justice goals (cf. [MR15]). If the environment fulfills the environment assumptions then the behavior of the generated controller meets the system guarantees. The winning condition is formulated as an implication between the liveness goals of the environment player and the liveness goals of the system player. Using a LEGO forklift controller as an example, Maoz and Ringert show that 52 out of the 55 specification patterns proposed in [DAC99] can be used to express assumptions and guarantees in the GR(1) fragment of LTL (cf. [MR15]). In [MR], GR(1) is used as the kernel of the Spectra language, which is introduced in [MR] as a specification language for reactive systems. The 52 LTL specification patterns, translated previously in [MR15] in the GR(1) fragment of LTL, constitute the catalog of patterns associated with language Spectra, to which user-defined patterns can be also added. The language Spectra is accompanied by Spectra Tools, which is a set of analysis tools, including a controller synthesis tool to obtain correct-by-construction system controllers (cf. [MR]).

GR(1) synthesis is also applied to extract controllers from scenario-based specifications. The work in [MS12] introduces A/G scenarios, which extend live sequence charts (LSCs) with support for environment assumptions (cf. [MS12]). Live sequence charts are a visual formalism used for the scenario-based modeling of system behavior. They represent an extension of message sequence diagrams by universal and existential modalities (cf. [MS12]), which allow the specification of mandatory, possible, and forbidden scenarios (cf. [HM08]). A/G scenarios can express safety assumptions, i.e., what the environment is assumed never to do, as well as liveness assumptions, i.e., what the environment is assumed to eventually always do. The semantics of the A/G scenarios is defined in terms of GR(1) and a game-based synthesis algorithm is used to generate a correct-by-construction controller (cf. [MS12]). The extension of LSCs through A/G scenarios does not introduce new constructs in the LSCs language, instead it embeds assumptions implicitly in the scenario definition (cf. [MS12]).

Greenyer [Gre11] extends modal sequence diagrams to specify real-time requirements and environment assumptions, in order to find inconsistencies in scenario-based specifications of mechatronic systems. The problem of finding inconsistencies in modal sequence diagrams is mapped to the problem of synthesizing wining strategies in two-player games (cf. [Gre11]). Assumptions are specified by assumption modal sequence diagrams, which are annotated with the respective stereotype defined in an UML profile specific for modal sequence diagram specifications. The modal sequence diagrams are mapped to a network of timed game automata that are used with Uppal Tiga for the purpose of controller synthesis (cf. [Gre11]).

6.2.3. Combining Design-time Verification with Runtime Validation

In [MGVP17], Mitsch et al. verify models of autonomous robots controllers against four different safety properties - static safety, passive safety, passive friendly safety, and passive orientation safety - using the theorem prover Keymaera X (cf. Section **??**). The model of the robot controller is formalized in dL. In addition to the design-time verification carried out with Keymaera X, the authors use ModelPlex [MP16], a feature of Keymaera X, in order to synthesize runtime monitors. The generated monitors include assumptions about obstacles as well as on the evolution domain of the hybrid dynamics and on the robot's decision on the next maneuver, e.g., braking, accelerating or staying at rest (cf. [MGVP17]). The synthesized monitors are then used to verify the compliance between the robot's implementation and the verified model of the robot's controller (cf. [MGVP17]). The runtime monitors synthesized with ModelPlex are integrated in the verification pipeline VeriPhy (cf. [BTM$^+$18]).

In [MP18], a controller in a train control system is verified offline against an explicit model of physics. The verified model is used to synthesize online monitors that check at runtime the implementation of the controller. The monitors synthesized from the physics model make assumptions on the model's continuous dynamics between sampling points (cf. [MP18]). This is done in order to account for partial observability due to uncertainty in the sensor data and partial controllability due to actuator disturbances (cf. [MP18]). The machine code implementation of the runtime monitors is obtained with VeriPhy (cf. [MP18]).

Desai et al. [DDS17] combine program model checking with runtime verification for validating the correctness of high-level robotic modules, that carry out the robot's discrete decision making and planing. The decision-making components and the planing components receive inputs from the robot's physical environment as well as from low-level closed-loop controllers, that implement the robot's continuous dynamics (cf. [DDS17]). The high-level decision-making and planing logic is modeled and verified with the event-driven programming language P. During model checking, the analysis carried out on the P modules uses assumptions to abstract the robot's continuous dynamics to a discrete behavior (cf. [DDS17]). The assumptions placed on the low-level controllers are formalized with the help of STL templates. The STL formula is learned from a set of robot trajectories and are instantiated with parameters inferred through regression analysis. The STL formula is then evaluated on an observed trajectory of the robot using an online monitor (cf. [DDS17]).

Ferrando [Fer19] uses model checking to verify runtime monitors at design-time against desired LTL properties, before using them to monitor the system during its operation. Often, runtime monitors are generated from LTL specifications. Rather than LTL properties, Ferrando uses trace expressions to construct the runtime monitors (cf. [Fer19]). Trace expressions represent a formalism more expressive than LTL, developed especially for the runtime verification of multi-agent system (cf. [AFM16]). In contrast to monitors constructed from trace expressions, a monitor generated from an LTL property satisfies the respective property by construction (cf. [Fer19]). In order to ensure that runtime monitors built from trace expressions satisfy the desired LTL properties, Ferrando generates an over-approximation of the trace expression, which is then verified against the LTL property using the Spin model checker (cf. [Fer19]).

In [CTT19a], Cimatti et al. use runtime verification to monitor partially observable systems. Assumptions modeled as fair transition systems are placed on the unobservable part of the system under analysis. Such an assumption can be a detailed model of the system under analysis or a constraint on a set of boolean variables (cf. [CTT19a]). The authors use the NuXmv model checker [CCD$^+$14] to synthesize a runtime monitor from the LTL property to be verified, using as inputs also the assumptions placed on the unobservable parts of the system (cf. [CTT19a]). The output of the runtime monitor depends on the information extracted from the partial observations of the system. The synthesized monitor has therefore a four-value semantic: (1) the LTL property is satisfied under the given assumption, (2) the LTL property is disproved under the given assumption, (3) the assumption is violated by the system under analysis, and (4) the result of the monitoring is inconclusive (cf. [CTT19a]). The approach is implemented in the tool NuRV, which extends the NuXmv model checker for runtime verification (cf. [CTT19b]). The system under analysis is a factory assembly line system specified in LTL (cf. [CTT19a]). The LTL properties considered for monitoring are the LTL specification templates defined in Dwyer's catalog of specification patterns in [DAC99] (cf. [CTT19a]).

In [HL19], Haupt and Liggesmeyer introduce an approach to derive rule-based safety monitors in order to observe the system state in adaptable autonomous systems and trigger mitigating actions, in case any safety violations are detected. The safety rules are derived through a HARA analysis, which takes into consideration the operational modes of the system and the system configurations associated with each of its operational modes (cf. [HL19]). An automation system used in agricultural machines serves as an example system for the application of this approach (cf. [HL19]). In [BHL], the authors present an approach for context-aware safety assurance of autonomous vehicles. They use a meta-model to differentiate between the system under analysis and the elements in its context. Furthermore, HARA analysis is applied to extract safety-critical parameters that can be used to define safety rules for the system monitoring during its operation (cf. [BHL]).

Dinh and Holvoet [DH20b] propose a framework to verify the decision making of robotic agents that operate in open environments. The item under analysis is the decision-making component of an unmanned air vehicle for pylon inspection (cf. [DH20b]). This component interacts with the agent's physical environment as well as with other technical components of the robot, e.g., on one side sensors and perception algorithms, and on the other side, motion planing, control algorithms and actuators (cf. [DH20b]). The perception algorithm is considered to produce continuous data and a discrete abstraction is applied on its output so that this output can be used by the robot's decision-making component (cf. [DH20b]). Rather than building an environment model for it, the authors specify the environment of this component as a set of LTL assumptions over the data received by the decision-making component from the perception component before the discrete abstraction is applied on it (cf. [DH20b]). The authors use the NuXmv model checker to verify the robot's decision-making component against system properties expressed in the property specification language of NuXmv, which is LTL with past and future operators (cf. [DH20b]). The LTL assumptions are derived on the basis of the authors' domain knowledge and the counterexamples produced during the verification with the NuXmv model checker (cf. [DH20b]). This study is extended in [DH20a] with other model checkers than NuXmv, e.g., ProB, Spin, and Alloy.

Reich and Trapp [RT20] introduce in their vision paper the framework Sinadra for situation-aware dynamic risk assessment. The framework combines design-time and runtime methods in order to build runtime monitors for situation-aware autonomous vehicles. At design-time, the Sinadra framework proposes to build a risk causality model through a risk and causality analysis that uses as inputs the vehicle's ODD specification and the behavioral models of the autonomous driving function (cf. [RT20]). The authors propose to transform the risk causality model into a Bayesian network model and use the latter at design-time to build a safety argument to show that the risk in specified ODD is not underestimated. The tables of conditional probabilities in the Bayesian risk model use as parameters the probability distributions of the situation factors defined in the vehicle's ODD specification (cf. [RT20]). At runtime, the nominal driving function perceives the performance-relevant of the environment and produces the intended ego vehicle behavior. The risk-relevant features are observed by the Sinadra monitors, which estimate the perception uncertainty for each risk-relevant feature (cf. [RT20]). The monitors use the Bayesian risk model and the estimated perception uncertainties in order to check the intended behavior of the vehicle for residual risk. The estimated residual risk is then given as input to a decision-making component, which provides the corresponding risk control measure (cf. [RT20]).

A similar approach is presented in [RSS$^+$20] for engineering of runtime safety monitors for cyber-physical systems. The notion of the proposed engineering framework relies on the notion of dependability digital identity (DDI) (cf. [RSS$^+$20]). This is defined as an integrated set of data models created at design-time by system engineers, that are used to build dependability arguments for the system under analysis (cf. [RSS$^+$20]). A DDI is formed as a modular assurance case, and contains claims about the dependability guarantees given by the system under analysis to other systems, supporting evidence for the dependability guarantees in the form of various models as well as demands from the other systems connected with the system under analysis, that are necessary to support the claims (cf. [RSS$^+$20]). The data models of a DDI serve also as a basis to build conditional safety certificates as a runtime assurance method for the system (cf. [RSS$^+$20]). Conditional safety certificates define failures modes together the respective variable bounds, context-specific constraints for which the variable bound must be valid, and a confidence level required for the assurance of the variable bounds (cf. [RSS$^+$20]). Conditional safety certificates support information abstraction and use boolean logic to express the safety guarantees and the safety demands for the system under test. The approach is demonstrated for a cooperative vehicle platooning function (cf. [RSS$^+$20]).

In their position paper, Ruchkin et al. [RCSL21] propose an approach for computing the overall assurance measure with which a learning-based autonomous system carries out its mission successfully. The proposed approach leverages design-time guarantees, by identifying what assumptions underlie these guarantees and develop runtime monitors that compute probabilistic confidence measures for these assumptions (cf. [RCSL21]). The system under analysis, an autonomous underwater vehicle in charge of pipeline inspections, and its environment are modeled as a hybrid system. The design-time verification is carried out in the form of reachability analysis with the tool Verisig (cf. [RCSL21]). The assumptions formulated in the example refer to the vehicle dynamics and the perception module, which is implemented with neural networks, e.g., the system's perception module provides accurate readings up to a predefined error bound (cf. [RCSL21]). In order to establish the relation between the assumptions violations and the violation of the system requirements, the authors carry out an assumption effect analysis, a technique which is considered similar to HARA (cf. [RCSL21]). Furthermore, they propose to use probabilistic analysis to derive the confidence output of the assumptions monitors from the observed behavior of the system components. The confidence measures computed by the individual assumptions monitors are given as input to the overall assurance monitor. The latter computes the assurance measure that the system fulfills its mission successfully (cf. [RCSL21]).

Watanabe et al. [WKLS18] integrate runtime monitoring in a development workflow in order to improve the safety of a vehicle during development and deployment stages.

The proposed development workflow starts with a set of requirements that the vehicle system under development must be designed to satisfy. Based on the system requirements the engineer constructs contracts that formally specify guarantees for the components behavior to be implemented and the assumptions on the environment (cf. [WKLS18]). The authors in [WKLS18] propose automatic synthesis to generate a runtime monitor from the specified contracts. The runtime monitors has two goals: (1) detection of potential faults during component test that may lead to a violation of the component's guarantee and (2) ensuring that the system components do not violate their assumptions during system integration (cf. [WKLS18]). The approach is applied for two automotive functions, a cooperative pile-up mitigation system and a false-start prevention system. The contracts are formalized in STL, the runtime monitor is generated as a C++ program or a MATLAB S-function, and the monitoring tool BREACH [Don10] is used to carry out the runtime monitoring (cf. [WKLS18]).

6.3. Comparison of the Proposed Approach with Related Work

This section compares the approach presented in this thesis with the related work reviewed in Section 6.1 and Section 6.2.

6.3.1. Comparison with the Manual Methods for Environment Assumptions Specification

This section takes a look at the manual methods for environment assumptions specification presented in Section 6.1.1 and compares them with approach introduced in this thesis.

Similar to this thesis, Arts et al. [ADT14] recognize the need to make explicit assumptions for the verification of safety requirements in safety-critical systems and use A/G contracts to achieve this. The A/G contracts used to verify the communication protection mechanism defined in the AUTOSAR standard are non-probabilistic and are used only for the design-time verification of the protection mechanism (cf. [ADT14]). In contrast, this thesis uses the environments assumptions in the design-time verification of the system's safety requirements (cf. Section 4.5), and then as basis to create runtime monitors that are used during system test (cf. Section 4.7) and requirements validation (cf. Section 4.8).

The probabilistic contracts proposed in [DCL11] allow computational model with a more restricted level of nondeterminism than an MDP because it assumes a unique global distribution for the choice of the next system state. By comparison, the safety-critical systems analyzed in this thesis through design-time verification methods are modeled as full MDPs (cf. Section 4.5). Li [LNSV^{+}17] and Nuzzo [NLSV^{+}19] focus on the verification of StSTL A/G probabilistic contracts formulated as a mixed integer optimization problem. In contrast, this thesis uses probabilistic model checking to verify probabilistic safety requirements under probabilistic environment assumptions formalized in PCTL (cf. Section 4.4 and Section 4.5). Furthermore, this thesis integrates the explicit specification of environment assumptions in a full-fledged system development process (cf. Section 4.1), while Li [LNSV$^+$17] and Nuzzo [NLSV$^+$19] focus on leveraging probabilistic A/G contracts for design-time verification and controller synthesis.

Gleirscher [Gle14] formulates A/G safety goals in PCTL*, yet in contrast to this thesis, does not verify them explicitly. In [GFN19], the theorem prover Isabelle/Hol is used to verify a mobile ground robot with respect to A/G pairs. Although a lot can be automated, theorem proving is still a semi-automated method for checking the correctness of a system with respect to given property (cf. Section 2.5.2). User input is required to guide the formal proof, e.g., by adding or removing axioms from the proof (cf. Section 2.5.2). In contrast, this thesis applies model checking to verify safety-critical systems with respect to their safety properties under consideration of explicitly defined environment assumptions. The A/G safety goals in [Gle14] are derived through safety analysis, similar to this thesis which uses safety analysis to derive both the safety requirements (cf. Section 3.4) and the environment assumptions (cf. Section 4.4). Similar to this thesis, Gleirscher's approach to the discovery of the A/G safety goals is integrated with the system development process and uses the artifacts obtained during the phases of system design and safety analysis (cf. [Gle14]). Nevertheless, Gleirscher's approach stays primarily on the left side of the V-model and does not address the phases of system test and requirements validation (cf. [Gle14]). By comparison, the approach presented this thesis is integrated with the full development process starting from the requirements analysis and safety analysis, continuing with the system design, system implementation and system test, and ending with the requirements validation (cf. Section 4.1).

The verification approaches for agent-based systems presented in [FDW13] and [DFL$^+$16] use BDI logic to formulate the assumptions of an agent about its environment. In [FDA$^+$18b] the assumptions of the agent about its environment are part of the environment model itself. The environment model is then used both for the verification of the agent via model checking as well as to generate runtime monitors, that validate the assumptions made in the environment model against the real environment of the agent. In [CFL$^+$20], the agent-environment interface is modeled with the Dafny tool, verified at design-time with the Z3 theorem prover and checked at runtime with the ROSMonitoring framework, with monitors synthesized from properties written in the RML language. In contrast to the works in [FDW13], [DFL$^+$16], [FDA$^+$18b], and [CFL$^+$20], this thesis proposes a concept that can handle verification and validation of both non-probabilistic and probabilistic environment assumptions. In this thesis, the environment assumptions are explicitly defined as a result of safety analysis, and not considered part of the environment model as in [FDA$^+$18b]. In [IQV16], changes in sensory perception and action feedback are provided with certain probability distributions, while the rational agent is modeled as a DTMC or an MDP, and verified with the Prism model checker. In contrast to this thesis, the work in [IQV16] neither defines explicitly any environment assumptions nor does it show how the specification of environment assumptions may be integrated in a comprehensive system development process.

6.3.2. Comparison with Automated Methods for Obtaining Environment Assumptions

This section takes a look at the manual methods for environment assumptions specification presented in Section 6.1.2 and compares them with approach introduced in this thesis.

Rather than having environment assumptions derived manually through the HARA analysis as it is done in this thesis, the learning-based approaches in [CGP03]. [GPB05], [PG06], [GGP07], and [PGGB$^+$08] use the learning algorithm L* and model checking to generate assumptions for non-probabilistic systems. The learned assumptions as well as the guarantees are deterministic finite automata. The AGAR approach discussed in [GBPG08] does not require the generated assumptions to be deterministic, which in turn results in smaller assumptions. These learning approaches are adapted for learning assumptions in asynchronous probabilistic systems (cf. [FKP10]) and for synchronous probabilistic systems (cf. [FHKP11]). In [Fen14], assumptions are learned for probabilistic systems whose components are DTMCs and are encoded as boolean formulae. The assumptions learned for probabilistic systems take the form of deterministic finite automata (cf. [FKP10]), probabilistic finite automata (cf. [FHKP11]), interval DTMCs (cf. [Fen14]), or labeled probabilistic transition systems (cf. [KPC12]). From the perspective of the system development process, learning-based methods can be mapped to the phase of system design. System engineers need to provide an environment model with which the technical system model can interact and a system requirement to be verified, even before environment assumptions can be automatically generated.

Assumptions are also generated via specification mining methods such as the ones presented in [LDS11], [Li14], [LSSS14], and [Ses19]. These methods take a property specification that is not realizable and attempt to synthesize a correct-by-construction controller from it by adding additional assumptions to an under-constrained environment of the system. From the perspective of the system development process, these methods are a combination between the phases of system design and system implementation.

It is worth noticing that the learning-based approaches and the specification mining methods presented in Section 6.1.2 are not integrated in the full-fledged system development process. In fact, these methods can be mapped either to the phase of system design, as is the case with the learning-based approaches for assumptions generation, or are situated at the intersection between the phases of system design and system implementation, as in the case of specification mining methods. Other phases of the system development process, requirements analysis, system test and requirements validation, are not taken into consideration. Furthermore, the assumptions obtained through the application of such methods are not formulated informally. Instead, these assumptions are specified in the respective formal language, e.g., GR(1) property specification or deterministic finite automata. As such, gaps may appear between the expectations of the system designers and those of other system stakeholders, e.g., end-users, since it is not expected that every system stakeholder has a working knowledge of the respective formal language.

6.3.3. Comparison with Methods that integrate Assumptions in Verification and Validation Processes

By comparison with [SKA13], this thesis uses model checking instead of theorem proving as design-time verification method. The safety requirement of the mobile service robot is comparable to the passive safety property introduced in [SKA13], which forbids the mobile service robot to actively collide with any obstacles in its environment. Furthermore, the safety requirement formulated for the mobile service robot is more comprehensive that the static safety property defined in [MGVP17], since it considers not only stationary but also dynamic obstacles in the robot's environment. Although the work in [MGVP17] introduces assumptions for each of the safety properties they analyze, their concept does not account for uncertainties in the system's environment that might be reflected in probabilistic assumptions. Furthermore, rather than defining them directly as in [MGVP17], this thesis illustrates how environment assumptions can be derived through safety analysis and be integrated in the system development process.

In [DH20b], a framework is proposed for the verification of decision-making components in robotic agents that operate in open environments. The decision-making component of the robot interacts with sensors and perception algorithms on one side as well as motion planing, control algorithms and actuators on the other side. The output of the perception algorithm is discretized so that it can be used by the decision-making component (cf. [DH20b]). The environment of the decision-making component is specified as a set of LTL assumptions over the data from the perception component before the discrete abstraction is applied on it (cf. [DH20b]). The assumptions are derived based on the author's domain knowledge and counterexamples produced during model checking of the decision-making component against the LTL system properties (cf. [DH20b]). By comparison with [**?**], this thesis derives environment assumptions through a HARA analysis. Furthermore, the concept of this thesis to allow the design-time verification of non-probabilistic systems as well as system that exhibit probabilistic and nondeterministic behavior under explicitly defined non-probabilistic as well as probabilistic environment assumptions.

Another work that sets out to make environment assumptions explicit is [DFL$^+$16]. In contrast with this thesis, the concept presented in [DFL$^+$16] does not consider uncertainty in the environment expressed via probabilistic assumptions. One scenario presented in [DFL$^+$16], adaptive cruise control, illustrates assumptions related to vehicles situated in physical environment of the ego vehicle (cf. [DFL$^+$16]). By comparison, this thesis also presents environment assumptions related to the dynamic behavior of obstacles in the physical environment of a mobile service robot (cf. Section 4.4). The other two scenarios presented in [DFL$^+$16], urban search and rescue and autonomous satellite, take into account the technical environment of the decision-making agent, which consists of other components with which the agent interacts, e.g., sensors, actuators, and planing component (cf. [DFL$^+$16]). In turn, this thesis considers environment assumptions which are related to the technical environment of the vehicle speed estimation function, e.g., longitudinal and lateral acceleration, as well as to the vehicle's physical environment, e.g., road gradient and GPS signal error (cf. Section 5.2.3). In [FDA$^+$18b], a continuation of the work in [DFL+16], runtime monitors are used to detect assumptions violations in the environment of rational agents. However, these assumptions are not explicitly defined, as in the present thesis. Instead, they are considered to be embedded in the environment models defined at design-time (cf. [?]).

Izzo et al. [IQV16] consider the probabilistic behavior of the environment of an autonomous surface vehicle. In contrast to this thesis, no explicit environment assumptions are made that are verified at design-time and validated at runtime. Instead, the probabilistic environment model is used to carry out design-time and runtime verification. Farrell [FCD+19] combines model checking and theorem proving to verify different software components in modular autonomous space robots. The concept presented in [FCD+19] follows the A/G reasoning approach, in which A/G properties are formulated for each software component of the system. However, the approach allows only for non-probabilistic assumptions, which are formulated in LTL. Furthermore, it is not clear where the assumptions come from and how are they integrated in the system development process.

In the synthesis approaches presented in [UPÇ12], [WUB+12], [WF12] and [WUB+13], [CKSW13], the knowledge about the environment behavior is encoded in the models used to represent the dynamic agents that exist in the system's environment. Although multi-objective model checking has been shown to be a good technique for compositional verification, especially for probabilistic assume-guarantee reasoning (cf. [FKNP11], [FKN+11]), the work in [CKSW13] does not explicitly specify assumptions about the adversarial environment in which the system operates. Though environment assumptions are formulated in [WTM12], the authors do not clarify how they obtain these assumptions, nor how their approach would integrate in a system development process such as the V-model. Furthermore it is worth noticing that, although explicitly defined, the environment assumptions are not explicitly and independently monitored. Instead, a GR(1) specification of the form $\psi_e \to \psi_s$ is verified with the specific goal to generate a new goal ψ'_s in case ψ_e is invalid.

The reactive synthesis approaches introduced in [MR15], [MR16], and [MR] transform LTL formulae from Dwyer's catalog of specification patterns [DAC99] to GR(1) formulae, which are then used to synthesize correct-by-construction controllers. The LTL patterns translated in GR(1) are associated with the language Spectra, which is a specification language for reactive systems introduced in [MR]. In contrast with this thesis, these approaches do not consider probabilistic assumptions, although the authors of [MR] claim that an adaptation of the Spectra language to accommodate probabilistic assumptions is possible. In [MS12], GR(1) synthesis is used to construct controllers from A/G scenarios. Assume-guarantee scenarios are introduced in [MS12] to extend live sequence charts with environment assumptions. The environment assumptions are not explicitly defined by specific constructs of the live sequence charts language, as it is done in this thesis. Instead they are embedded implicitly in the scenario definition (cf. [MS12]). Assumptions can be formulated also in graphical modeling languages. Greenyer specifies assumptions by assumptions modal sequence diagrams, in order to find inconsistencies in the scenario-based specification of mechatronic systems (cf. [Gre11]). Assumption modal sequence diagrams are identified through a specific UML stereotype. For the purpose of controller synthesis, the modal sequence diagrams are mapped to a network of timed game automata, which are then analyzed with Uppaal Tiga (cf. [Gre11]). In comparison to this thesis, the approach in [Gre11] does not allow for probabilistic assumptions.

Several works combine design-time verification and runtime validation to check the correctness of safety-critical systems. In [MP18], a controller in a train control system is verified offline against an explicit model of physics. From the verified physics model, online monitors are synthesized that check at runtime the implementation of the controller. The synthesized monitors make assumptions on the physics model's continuous dynamics between sampling points, in order to account for partial observability due to uncertainty in the sensor data and partial controllability due to actuator disturbances (cf. [MP18]). Compared to [MP18], this thesis considers assumptions on the behavior of obstacles in the environment of a mobile service robot (cf. Section 4.3 and Section 4.4) and sensor uncertainty with respect to GPS sensor data for a vehicle speed estimation function (cf. Section 5.2). Model checking are combined with runtime verification [DDS17] to validate the correctness of high-level robotic modules, that carry out the robot's discrete decision making and planing. Assumptions are made during model checking of the decision-making and planing logic in order to abstract the robot's continuous dynamics to a discrete behavior (cf. [DDS17]). Rather than formulating the assumptions directly in STL for the low-level controllers of the robot, these are learned from robot trajectories and parameterized with parameters obtained through regression analysis (cf. [DDS17]). By comparison with this thesis, the approach presented in [DDS17] does not account for uncertainty in the robot's environment, and as such does not allow probabilistic assumptions.

The work in [Fer19] focuses on gaining trust in the specification of a system before using that specification to monitor the system itself. For this purpose, runtime monitors obtained from trace expressions are verified at design-time via model checking against desired LTL properties, before being used to monitor the system during its operation (cf. [Fer19]). However, this thesis brings forth the argumentation that environment assumptions are integral part of such a specification and must be explicitly defined and monitored in order to successfully verify the system during design-time and validate it during operation time. In [CTT19a], runtime verification is used to monitor partially observable systems. Assumptions in the form of fair transition systems are placed on the unobservable parts of the system. These assumptions are used as inputs in the generation of runtime monitors from LTL properties. In contrast to this thesis, assumptions in [CTT19a] may take the form of a detailed model of the system under analysis or a constraint on a set of boolean variables (cf. [CTT19a]). It is worth noticing that these assumptions are not explicitly monitored as it is done in this thesis, but are used as inputs for the generation of runtime monitors.

Haupt and Liggesmeyer [HL19] use rule-based safety monitors to observe the system state in and trigger mitigating actions, in case any safety violations are detected. The safety-rules are derived through a HARA analysis under consideration of the system context. By comparison with this thesis, the work in [HL19] does not consider environment assumptions explicitly neither during system design-time nor during runtime.

Furthermore, the proposed approach does not make any quantitative provisions for the uncertainty which is inherent to contexts of autonomous systems.

Although the work in [RT20] uses design-time and runtime methods, it adopts a completely different approach in comparison to this thesis. In this thesis, environment assumptions are explicitly defined at design-time and used for the design-time verification of the system under analysis. At runtime, the environment assumptions are validated with the help of runtime monitors constructed from the explicit environment assumptions specification and tested during system test.

The approach presented in [RT20] aims to build probabilistic runtime risk monitors for situation-aware autonomous vehicles. At design-time, the authors propose to use risk causality models derived through a risk and causality analysis and modeled as Bayesian networks in order to build a safety argument that shows that the risk in the vehicle's ODD specification is not underestimated (cf. [RT20]). Runtime monitors use the Bayesian risk model and estimated perception uncertainties to check the intended behavior of the vehicle for residual risk (cf. [RT20]). In contrast, this thesis carries out design-time verification during system design in order to give a mathematical proof of the system correctness with respect to its safety property under explicitly defined environment assumptions. Furthermore, these assumptions are validated at runtime in order to guard the system against unforeseen situations.

In [RSS$^+$20], modular assurance cases are used at design-time to build dependability arguments for the system under analysis. These assurance cases contain the dependability guarantees given by the system under analysis to other systems, supporting evidence for the dependability guarantees in the form of various models, and demands from the other systems connected with the system under analysis, that are necessary to support the claims (cf. [RSS$^+$20]). Conditional safety certificates are used at runtime for the system validation and define failures modes together their variable bounds, context-specific constraints for the variable bounds, and a confidence level for the assurance of the variable bounds (cf. [RSS$^+$20]). In contrast to this thesis, the work in [RSS$^+$20] uses assurance cases and conditional safety certificates to give a method for the construction of safety cases for automated cyber-physical systems. It does not present a practical concept for the derivation and explicit definition of environment assumptions, their usage in design-time verification validation during system operation time, as it is done in this thesis.

In the position paper in [RCSL21], an approach is proposed that aims to identify what assumptions underlie the design-time guarantees of an autonomous underwater vehicle for pipeline inspection and develop runtime monitors that compute probabilistic confidence measures for these assumptions (cf. [RCSL21]). Although the authors formulate assumptions with respect to the vehicle dynamics and the perception module, these assumptions are vague and are not measurable. Furthermore, they do not provide a formal specification for these assumptions nor do they make explicit how these assumptions are used during the design-time verification with the reachability analysis tool Verisig. The authors propose to use probabilistic analysis to built assumptions monitors. However, it is not clear how this could work in the absence of a formal specification for probabilistic assumptions.

In [WKLS18], runtime monitoring is integrated in a development workflow to improve the safety of a vehicle during development and deployment stages. The authors use the system requirements to build A/G contracts that formally specify guarantees for the component's behavior to be implemented and the assumptions on the environment (cf. [WKLS18]). The authors in [WKLS18] propose automatic synthesis to generate a runtime monitor from the specified contracts. The runtime monitor has two goals: (1) detection of a violation of the component's guarantee and (2) detection of a violation of the component's assumptions during system integration (cf. [WKLS18]). The assumptions and the guarantees are formulated in STL. In comparison with this thesis, the work in [WKLS18] does not show how the assumptions can be used in design-time verification. Furthermore, their concept does not allow for probabilistic assumptions, which could be used to address the uncertainty inherent to the system environment.

6.4. Summary

This chapter introduced state-of-the-art research work related to the scope of this thesis. In order to do this, the chapter started by presenting methods for obtaining environment assumptions. Two distinct ways have been identified by which environment assumptions can be obtained: (1) direct manual specification by experts with domain knowledge or through learned-based methods or synthesis from observations in the system's environment. Further, this chapter discussed approaches that used environment assumptions in verification process at design-time or in validation process during system runtime. Furthermore, the chapter took a deeper look at research work that combined design-time verification and runtime validation at the role(s), if any, which environment assumptions play in the system development process. The chapter concluded with a thorough comparison between the RMEA approach introduced in this thesis and all the presented research work.

The conclusion gained from this chapter is that, although some approaches emphasized the necessity for an explicit definition of environment assumptions, none of the approaches presented in the research work has given a concept for the usage of environment assumptions that is general enough to be applicable both to probabilistic as well as non-probabilistic systems and at the same time practically integrated in a full-fledged system development process.

Chapter 7. Summary and Conclusion

7.1. Discussion of Results 257
7.1.1. Contributions 258
7.1.2. Limitations of this Thesis and Future Work 260
7.2. Summary 264

This chapter concludes this thesis by discussing in Section 7.1 the contributions brought forward by this thesis and the limitations of this work. Starting from these, the directions for future research are presented. Section 7.2 summarizes the whole thesis.

7.1. Discussion of Results

This section discusses the results obtained in this thesis. This discussion is divided in two parts: (1) discussion of the thesis contributions in Section 7.1.1 and (2) discussion of the limitations identified for the approach presented in this thesis as well as directions for future research in Section 7.1.2.

Chapter 3 analyzed the problem that is addressed in this work. As a result of the problem analysis, the general research question of this thesis was derived:

> *How can design-time guaranteed environment assumptions be used at runtime to continuously validate the design-time verification result with respect to autonomous safety-critical systems operating in uncertain environments?*

In order to provide an answer to it, the general research question was refined into three more precise research questions. These research questions were introduced in Section 3.10.2 and are reiterated in this section as a point of reference for the discussion regarding the contributions of this thesis in Section 7.1.1.

Research Question 1 (RQ-1) How can environment assumptions and their relation to the system's property specification be explicitly and formally described?

Research Question 2 (RQ-2) How can the formal description in RQ-1 be used to construct environment assumptions monitors?

Research Question 3 (RQ-3) How can the applicability of this approach be demonstrated for real-world safety-critical systems?

7.1.1. Contributions

The main contribution of this thesis is the RMEA approach. It is a safety engineering approach which combines the design-time verification and runtime validation of explicitly defined environment assumptions in order to ensure the safety of automated safety-critical systems.

In order to realize the RMEA approach, three main contributions are produced in this thesis, each in response to one of the resarch questions.

Method for the Explicit and Formal Definition of Environment Assumptions This method is integrated with the system development process and uses its specific phases and their corresponding artifacts in order give an explicit and formal definition of environment assumptions. Based on a high-level description of the system under development, the functional system requirements were defined during the phase of requirements elicitation and analysis. The same high-level system description was used to carry out the safety analysis and derive safety hazards that could appear during system operation. Then, safety requirements and environment assumptions were defined that cover the respective safety hazards. The environment assumptions were used to extend the system safety requirements, denoted from then on as extended safety requirements. A requirements pattern is designed and applied in order to produce the informal specification of the extended safety requirements. Their formal specification was realized through TCTL for non-probabilistic systems and using a fragment of PCTL for probabilistic systems. The functional system requirements and the extended safety requirements served then as an input for the design of the technical system model and the environment model. Using the environment assumptions, these models were verified at design-time against the system safety requirements. The result of the design-time verification showed that the technical system model satisfied the system safety requirements under the consideration of the explicitly defined environment assumptions. The method for the explicit and formal definition of environment assumptions is in response to RQ-1.

Method for the Construction of Environment Assumptions Runtime Monitors This method is rooted in the concept of the RMEA approach and takes the formal definition of environment assumptions as input and produces as output environment assumptions monitors. For the construction of environment assumptions monitors to succeed, it was important to understand how the explicit definition of environment assumptions on the requirements level was mirrored at system design level. Environment assumptions constitute the interface between the technical system model and the environment model. An analysis of this interface was carried out and the environment assumptions were mapped on the technical system model and the environment model. The mapping of the environment assumptions on the technical system model was then projected on the implemented system. Further variable definitions were introduced to model the observations of the environment assumptions monitors in the system environment.

The definition of the environment assumptions monitors combined variables of the implemented system and variables that modeled the monitor's observations in the system environment. On a concrete level, the monitors were defined as predicates in first-order logic. The method for the construction of environment assumptions monitors is in response to RQ-2.

Demonstration of the Applicability of the RMEA Concept on two Real-world Safety-critical Systems In order to demonstrate the applicability of the RMEA approach on real-world systems, this thesis presented two case studies built around (i) an autonomous mobile service robot commissioned to execute transportation tasks in a hospital environment and (ii) an automotive function which estimates and displays a moving vehicle's speed on its instrument board. For each system taken into consideration, the created artifacts follow the specific phases of the system development process, thus demonstrating the applicability of the RMEA concept on the respective system. The two case studies on which the applicability of the RMEA approach was demonstrated represent the response to RQ-3.

In addition to the high-level discussion of the main contributions of this thesis, it is worth giving a more detailed description of the contributions by discussing the artifacts and results produced for each research question.

This thesis addressed RQ-1 by developing a method for the explicit and formal definition of environment assumptions. The development of this method is aligned with the system development process. As such, the phases of the development process are constituting steps of this method and the artifacts obtained as a result of the respective development phases contribute to the final result of this method, i.e. the explicit formal definition of environment assumptions. The artifacts that resulted in each step of the method are highlighted in heavy print. The first step of this method is the requirements elicitation and analysis. The second step of this method is the safety analysis using HARA. These two steps share the same input, i.e., a high-level description of the system under development. The output of the requirements elicitation and analysis is a **catalog of functional system requirements**. The output of the HARA analysis is a **catalog of extended safety requirements**. An extended safety requirement consists of a safety requirement extended with explicitly defined environment assumptions. The environment assumptions cover specific safety hazards, that were identified through a HARA analysis. The **informal specification of the extended safety requirements** is formulated using as a tool a **requirements pattern** that was specifically design for this purpose. The **formal specification of the extended safety requirements** is created by using an **EBNF grammar** for the probability expression in the extended safety requirements and temporal logic for the clauses of the environment assumption and the main safety requirement. The environment assumptions were used for the design-time verification of the overall system model with respect to the system safety requirements. The overall system model consists of the parallel composition between the **technical system model** and **the environment model**. These models were created during system design on the basis of the functional system requirements and of the extended safety requirements. The design-time verification was carried out with the Prism model checker in the first case study and with the Storm model checker in the second case study. For the verification with Prism and respectively Storm, the extended safety requirements were formalized in a fragment of PCTL, a formal language supported by both model checkers.

In response to research question RQ-2, a method for the construction of runtime monitors for environment assumptions was developed. The method takes as input the formal specification of an explicitly defined environment assumption and produces as output the formal definition of the respective runtime monitor. The artifacts that resulted throughout the application of this method are highlighted in heavy print. The first step of this method is the analysis of the environment assumptions. This step is followed by the mapping of the environment assumptions on the technical system model and the environment model. Through this mapping, relevant variables are identified in the technical system model and in the environment model, that contribute to the definition of the environment assumptions. The variables identified in the technical system model are then mapped on the implemented system. This step is then followed by the definition of variables that model the observations of the environment assumptions monitor. In the final step of this method, the **formal definition of the environment assumptions monitors** is constructed using variables from the implemented system and variables introduced with the runtime monitor itself to model its observations of the system environment. For the purpose runtime validation, the environment assumptions monitors are defined in first-order logic.

In order to address research question RQ-3, two case studies were conducted. The goal of these case studies was to demonstrate the applicability of the RMEA concept on two different real-world safety-critical systems. The first case study was built around a mobile service robot which was commissioned to execute transportation tasks autonomously in a hospital environment. The second case study considered an automotive system function which estimates and display's the speed of a moving vehicle on its instrument board. As a result of the application of the RMEA concept, the following artifacts were produced for each case study:

1. Catalog of functional and safety requirements,
2. Catalog of safety hazards derived through HARA analysis,
3. Catalog of extended safety requirements that cover the respective safety hazards,
4. Design-time models of the system under analysis and its environment,
5. Runtime monitors for the environment assumptions, and
6. Test cases to test both the system under analysis and the environment assumptions.

7.1.2. Limitations of this Thesis and Future Work

The contributions of this thesis were introduced in the previous sections. In this section limitations of this thesis approach as well as opportunities for future research directions are discussed.

Limitations regarding the System Development Process

The V-model is the system development process that was used in this thesis. International standards recommend it for the development of safety-critical systems (cf. [Int11b], [Int19]). The premise of the V-model is that the system artifacts produced on the left branch of the system development process, e.g., system specification, use cases, and environment model, are completely documented and validated. This idea corresponds to the closed-world assumption. Nevertheless, in this thesis, automated safety-critical systems are considered to operate under the open-world assumption. This means that the system artifacts produced on the left side of the V-model, e.g., requirements catalog and system design models, are inherently incomplete.

Rather than using the classical V-model as a development process which relies on the closed-world assumption, it seems reasonable to use a system development process that operates under the open-world assumption. An engineering approach for dependable autonomous systems that operates under the open-world assumption is presented in [AHDR18]. The approach proposes developing and testing runtime monitors at design-time based on the system artifacts available in the current development iteration, and then deploying the runtime monitors alongside the tested system in the operational environment (cf. [AHDR18]). During operation, the runtime monitors collect data with respect to the system behavior, e.g., whether the system satisfied its current requirements or not. These data are logged and used in the following iteration of the system development in order to improve the system design artifacts (cf. [AHDR18]). Mauritz [Mau19] has already introduced a quantitative monitor in order to log the runtime data and has shown how this can be used to find new test cases for a lane keep assist system. However, the analysis of the data and the development of the test cases was done manually (cf. [Mau19]). One possible future research direction is automatizing the process of the runtime data analysis and the improvement of the system design artifacts.

Limitations regarding the Elicitation and Analysis of System Requirements

The system's functional requirements together with its safety requirements are used as input for the design of system models and for the design-time verification use as input the catalog of the functional system requirements. The underlying premise is that the functional system requirements are fully known, are complete, and are unambiguously specified (cf. [KW16]). In fact, requirements for an automated safety-critical system may change during the system development process several times, e.g., through the addition of new functionality to the system. The natural language in which system requirements are formulated is imprecise and ambiguous. Formal verification methods employed at design-time affords system designers the mathematical proof that the developed system model satisfies the specified system requirement. In order to carry out formal verification, the system requirement must be expressed in a formal language, e.g., PCTL. Although research works have been developed that try to bridge it, e.g., through property specification templates, requirements patterns, or the use of natural language processing methods, a semantic gap between system requirements formulated in natural language and their respective formal specification may still exists due to the requirements complexity for automated safety-critical systems. Using requirements patterns can help reduce the ambiguity of natural language. However, developing a requirement pattern that is able to encompass all system requirements is difficult and often depends on what the particularities of the system under analysis are. As an example, the requirements pattern developed in this thesis does not account for timing requirements.

Automated safety-critical systems often have components highly complex machine-learning algorithms, e.g., deep learning neural networks used to process large amounts of sensor data automated vehicles in order to create an environment model of the vehicle. Requirements for such components cannot be formulated in the classical way, using natural language. Instead, the training data gathered for training the neural networks could be considered as a requirement for the machine-learning component. However, explaining why a particular set of training data is a good requirement for a given system component is challenging without further knowledge of what the rest of the system does.

Future research work can be aimed at the development of methods to automatically extract new requirements from the data logged during system operation, and using the new requirements to improve and make the system requirements complete. Runtime monitors can be used to comb through the data logged during system operation and detect any emergent system behavior, from which then new requirements specification can be extracted via specification mining methods (cf. [AAK$^+$22]). Specification completion methods must be also developed that can take the mined requirements specification as input and use it to complete the existent requirements specification of the system.

Limitations regarding the Derivation of Environment Assumptions

In this thesis, environment assumptions are derived the same as safety requirements through the HARA analysis. The purpose of the safety analysis is to identify possible safety hazards for the automated safety-critical systems, by analyzing the defined system behavior as well as possible changes in the environment factors. However, any safety analysis is inherently incomplete. The incomplete set of hazards identified through the safety analysis leads to incomplete set of environment assumptions.

Future work in this area relates to addressing the incompleteness of environment assumptions by observing the behavior of the other environment agents and use specification mining techniques to extract environment assumptions from it. Various learning-based methods and specification mining techniques have been discussed in Section 6.1.2. Learned-based methods rely on a combination of the L* algorithm and model checking in order to learn assumptions in an iterative manner. Specification mining methods generate additional assumptions in order to make an original property specification realizable through the implementation of a controller. Since a specification is realizable if a correct-by-construction controller can be synthesized from it, specification mining methods deal with some of the challenges in controller synthesis, e.g., the complexity of the synthesis algorithm which is double exponential in the length of the formula in case of LTL specifications (cf. [MR15]).

Limitations regarding System Design and System Implementation

The technical system model and the environment model were created manually during design-time based on the specification of the functional system requirements and the specification of the safety requirements extended with environment assumptions. For more complex automated safety-critical systems, there is a gap between the system design model and the requirements. The system was implemented manually. For complex safety-critical systems manual implementation may prove cumbersome as the challenge is to ensure consistency of the system implementation with the system design models. Qualified code generators, e.g., Kcg of the Ansys Scade toolchain, have proven successful in addressing this problem for engineered systems. Another possibility is to use reactive synthesis in order to generate correct-by-construction from formal property specifications (cf. [MR15], [MR15]). In this way, the phase of designing system model is bypassed, because the synthesized controller is both its own model and its implementation. The challenge of reactive synthesis is the computational complexity of the synthesis algorithm which depends on the length of the formal specification formula (cf. [MR15]).

Future research directions in regard of system design and system implementation are to investigate methods for controller synthesis from probabilistic requirements under probabilistic assumptions. Notice that, in case of probabilistic systems, learning-based methods may not yield any assumptions that can be used to make a probabilistic system requirement realizable through a controller implementation.

The environment assumptions monitors developed in this thesis can be viewed as dependability cages for the system environment, however without the component for fail-operation reaction defined in the architecture of the dependability cage (cf. [AHDR18], [HABR22]). Future research work regarding the development of the environment assumptions monitors can concentrate on the adaptation of the monitors so that data can be logged during operation time, particularly data in which violations of the environment assumptions are visible. Furthermore, environment assumption monitors could be applied in future research to detect whether the automated safety-critical system is still operating in the boundaries of its ODD or not.

Limitations regarding System Test and Requirements Validation

Test cases are developed in this thesis with respect to the safety requirements can be complete only with respect to the set of system requirements and the set of environment assumptions. Edge test cases are difficult to identify systematically identified due to the incompleteness of the system requirements and of the environment assumptions.

Future work regarding system test should address methods to enrich the system test suite with new unique test cases. One possibility is to expand specification mining techniques with test case generation techniques in order to obtain new test cases from new system requirements and environment assumptions learned from runtime logs recorded during system operation.

During requirements validation, the environment assumptions defined for the system under analysis are tested in the operational environment via the environment assumptions monitors. The success of the requirements validation depends heavily on the performance of the sensors. Perception systems may have difficulties detecting color variations due to shadows. Large reflective surfaces may make determining object positions or even recognizing the object at all very difficult. Delay in receiving sensor data can also affect the performance during requirements validation. The requirements validation of the mobile service robot took place in a hospital ward, in which very few reflective surfaces existed so that the performance of its ultrasound sensor array and its laser scanner was essentially not affected. In case of the automotive speed estimation function, the requirements validation did not take place due to a pending preliminary review from German certification authority.

Future work with respect to requirements validation entails running extensive field tests in different operational environments with changing characteristics.

7.2. Summary

This chapter has summarized this thesis by outlining the main contributions, limitations and future work.

Due to their safety-critical nature, automated safety-critical systems benefit from verification and validation methods that ensure the safety of the system. Formal verification methods make use at design-time of formal system models in order prove the system correctness with respect the system safety property specification. When designing the system model, system engineers rely often on several implicit assumptions about the system environment which are used in the computations of the system model. The result of design-time verification methods is only as good as the model and some the methods suffer from scalability problems, e.g., model checking, or are only partially automated, e.g., theorem proving. Testing is used to complement the design-time verification and check the system against the system requirements via test oracles. During system operation, sfety hazards may appear not due to the malfunctioning of the system, but rather because of unforeseen events that occur in the operational environment of the system. These events invalidate the environment assumptions used by system designers during design-time. Since the environment assumptions are not explicitly defined, there is no way for the system under test to be aware that its environment assumptions have become invalid.

The RMEA approach is a safety engineering approach which proposes that environment assumptions are explicitly defined and formally specified at design-time and then validated during system operation. The approach develops a method for the explicit and formal definition of environment assumptions which is integrated with the system development process and uses its specific phases and their corresponding artifacts in order obtain a formal specification of environment assumptions. The formal specification of the environment assumptions is used for the definition of environment assumptions monitors. The applicability of the approach has been demonstrated on two real-world automated safety-critical systems: (1) a mobile service robot used for transportation tasks on a hospital environment, and (2) an automotive function for the accurate vehicle speed estimation.

Appendix A.

Addendum to Case Study 2: Vehicle Speed Estimation Function

A.1. Design-time Verification of Extended Safety Requirements ESR2 - ESR4

This appendix shows how the design-time verification of the speed estimation function works for the extended safety requirements ESR2, ESR3, and ESR4, that have been formally specified in Table 5.4. The extended safety requirements are translated as a multi-objective property in PCTL and verified with the STORM model checker. Table A.1 shows each multi-objective property alongside the corresponding formal specification of the extended safety requirement, together with its verification results. The translation to multi-objective property of ESR1 and the respective design-time verification results is shown in Section 5.2.4.

Table A.1.: Vehicle Speed Estimation Function - Extended Safety Requirements ESR2 - ESR4 expressed as Multi-objective Properties.

ID	Formal Specification	Prism Multi-objective Properties	Verification Result
ESR2	$P_{\geq 0.99}(G\ [\alpha_{Road} < 12\%]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$	`multi(P>= 0.99 [G (roadGradient <= 0.12)], P>=0.95 [G ((0 <= vDisplay - vReal) & (vDisplay - vReal <= 5))])`	✓
ESR3	$P_{\geq 0.99}(G\ [a_{Long} \leq 0.001]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$	`multi(P>= 0.99 [G (aLong <= 0.001)], P>=0.95 [G ((0 <= vDisplay - vReal) & (vDisplay - vReal <= 5))])`	✓
ESR4	$P_{\geq 0.99}(G\ [a_{Lat} \leq 0.0001]) \rightarrow P_{\geq 0.95}(G\ [(0 \leq v_{Display} - v_{Real}) \wedge (v_{Display} - v_{Real} \leq 5)])$	`multi(P>= 0.99 [G (aLat <= 0.0001)], P>=0.95 [G ((0 <= vDisplay - vReal) & (vDisplay - vReal <= 5))])`	✓

A.2. Definition of Runtime Monitors for the Environment Assumptions of ESR2 - ESR4

For each environment assumption in the extended safety requirements ESR2 - ESR4 specified in the Table 5.4, the respective monitor is defined according to the Definition 4.5.1 (cf. Section 4.5.5). Equations (A.1) to (A.3) show the monitors defined for the environment assumptions regarding the road slope, the longitudinal acceleration, and the lateral acceleration of the vehicle.

$$M_{\psi_{RoadSlope}} : G\ (f_{Observed}^{RoadSlope} \geq p_{Assumed}^{RoadSlope}) \tag{A.1}$$

$$M_{\psi_{LongAccel}} : G\ (f_{Observed}^{LongAccel} \geq p_{Assumed}^{LongAccel}) \tag{A.2}$$

$$M_{\psi_{LatAccel}} : G\ (f_{Observed}^{LatAccel} \geq p_{Assumed}^{LatAccel}) \tag{A.3}$$

where $p_{Assumed}^{RoadSlope} = 0.99$, $p_{Assumed}^{LongAccel} = 0.99$, and $p_{Assumed}^{LatAccel} = 0.99$ are the assumed probabilities of occurrence specified for the events $\psi_{RoadSlope}$, $\psi_{LongAccel}$, and $\psi_{LatAccel}$ in ESR2, ESR3, and respectively ESR4.

The events $\psi_{RoadSlope}$, $\psi_{LongAccel}$, and $\psi_{LatAccel}$ are defined in Equation (A.4), Equation (A.5), and respectively Equation (A.6):

$$\psi_{RoadSlope} \overset{\text{def}}{=} (\alpha_{Road} \leq 0.12) \tag{A.4}$$

$$\psi_{LongAccel} \overset{\text{def}}{=} (a_{Long} \leq 0.001) \tag{A.5}$$

$$\psi_{LatAccel} \overset{\text{def}}{=} (a_{Lat} \leq 0.0001) \tag{A.6}$$

The relative frequency of occurrence for each of the three events is computed as shown in Equations (A.7) to (A.9):

$$f_{Observed}^{RoadSlope} = \frac{n_{Observed}^{RoadSlope}}{N} \tag{A.7}$$

$$f_{Observed}^{LongAccel} = \frac{n_{Observed}^{LongAccel}}{N} \tag{A.8}$$

$$f_{Observed}^{LatAccel} = \frac{n_{Observed}^{LatAccel}}{N} \tag{A.9}$$

In Equations (A.7) to (A.9), N represents the total number of trials or events taken into consideration, while $n_{Observed}^{RoadSlope}$, $n_{Observed}^{LongAccel}$, and $n_{Observed}^{LatAccel}$ represent the absolute frequencies of occurrence for the events $\psi_{RoadSlope}$, $\psi_{LongAccel}$, and respectively $\psi_{LatAccel}$.

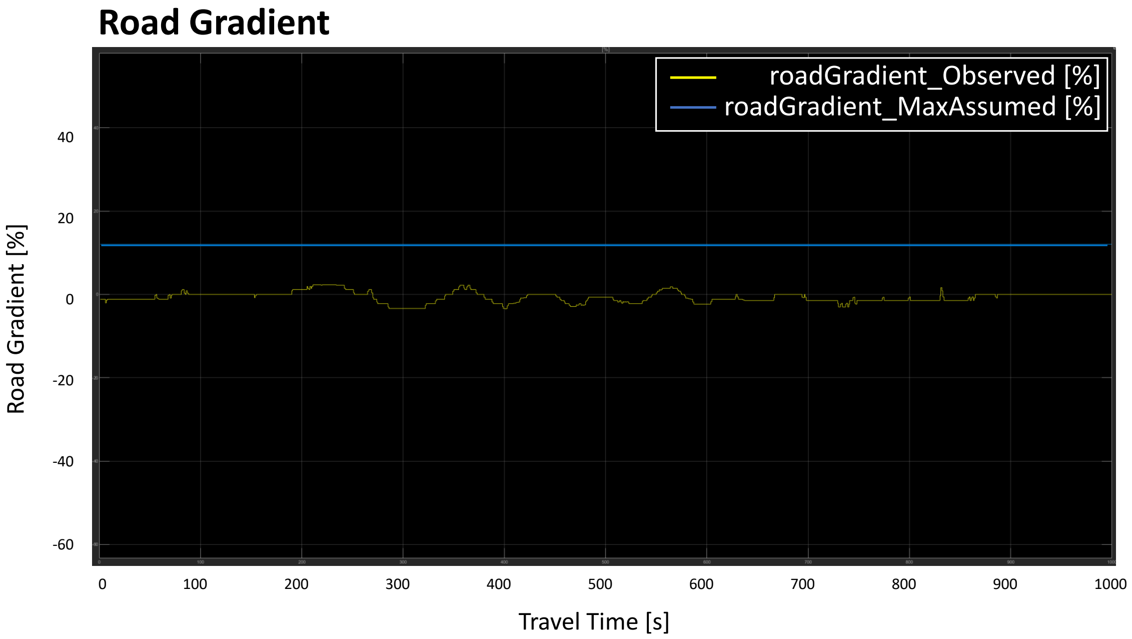


Figure A.1.: Vehicle Speed Estimation Function: Environment Assumption Monitor for the Road Slope during the *Smooth Driving* Scenario.

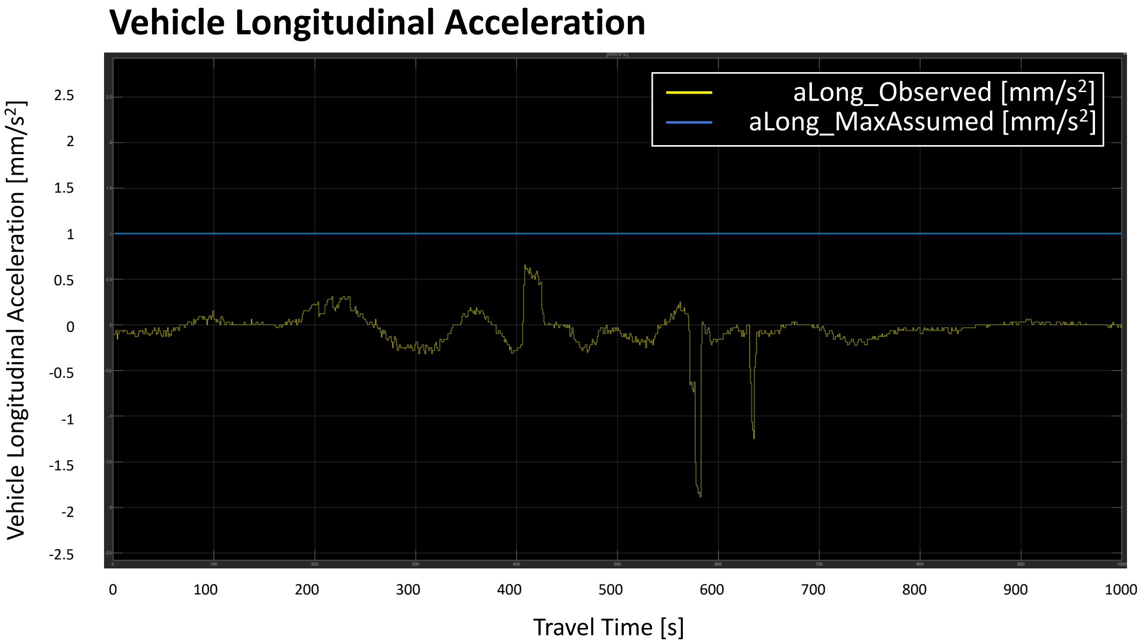
A.3. System Test of the Environment Assumptions Monitors of ESR2 - ESR4

This section presents the evaluation of the monitor traces for the environment assumption monitors regarding the road gradient, $M_{\psi_{RoadSlope}}$, the longitudinal acceleration $M_{\psi_{LongAccel}}$, as well as the monitor for the lateral acceleration $M_{\psi_{LatAccel}}$.

A.3.1. Scenario 1: Smooth Driving

Figure A.1 depicts the execution of the environment assumptions monitor with respect to the road gradient during system test. Figure A.2 shows the execution of the environment assumptions monitors with respect to longitudinal and latitudinal acceleration during system test. In each of the depicted graphics the yellow curve represents the observed values, while the blue curve represents the maximum assumed value in the respective monitor.

Notice that the environment assumptions of the road gradient, the longitudinal acceleration and the lateral acceleration are valid because the values observed by the respective environment assumptions monitors are distinctly situated under the maximum assumed value throughout the traveling time of the ego vehicle.



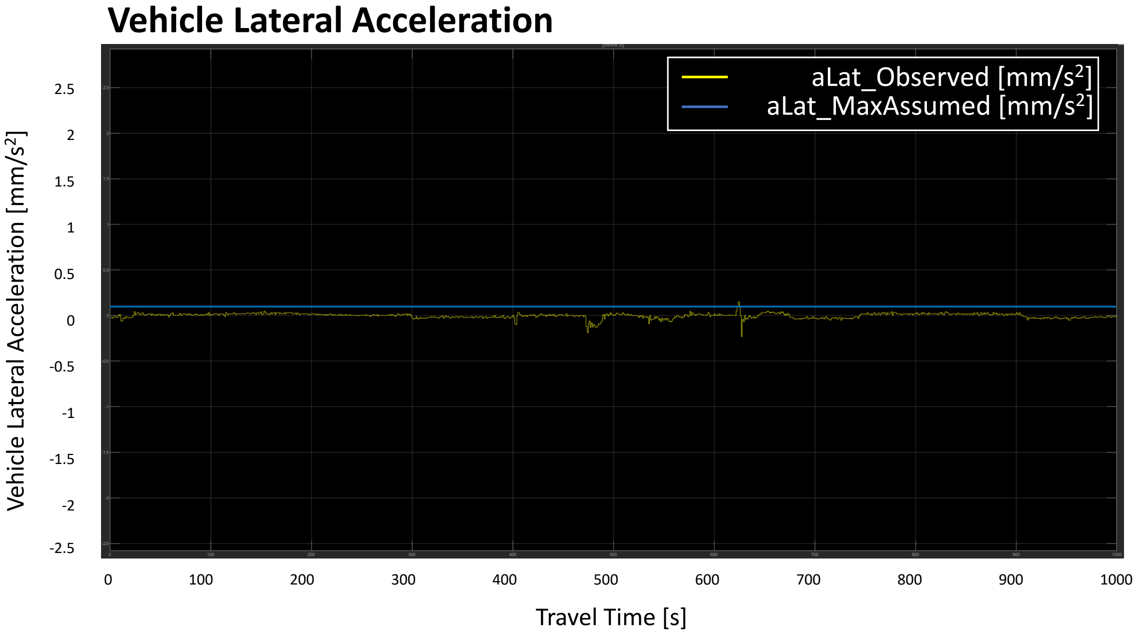


Figure A.2.: Vehicle Speed Estimation Function: Environment Assumptions Monitors for the Longitudinal Acceleration and the Lateral Acceleration during *Smooth Driving* Scenario.

A.3.2. Scenario 2: Dynamic Driving

Figure A.3 illustrates the monitor traces for the road gradient environment assumption monitor, while Figure A.4 shows the monitor traces for the longitudinal acceleration and lateral acceleration of the vehicle logged during the dynamic driving scenario in system test.

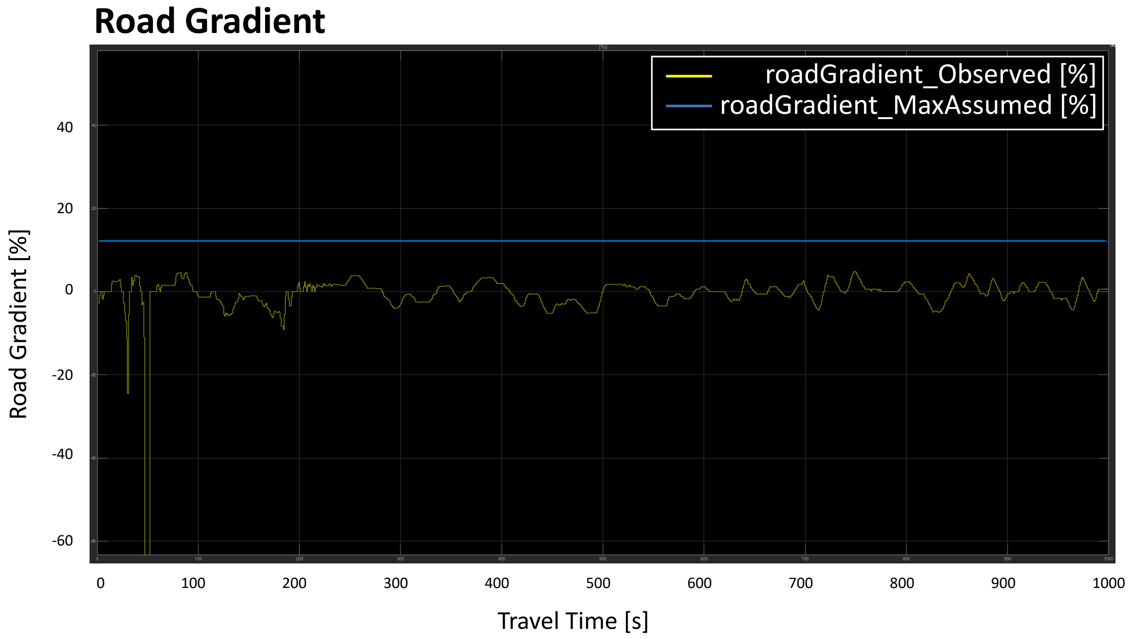
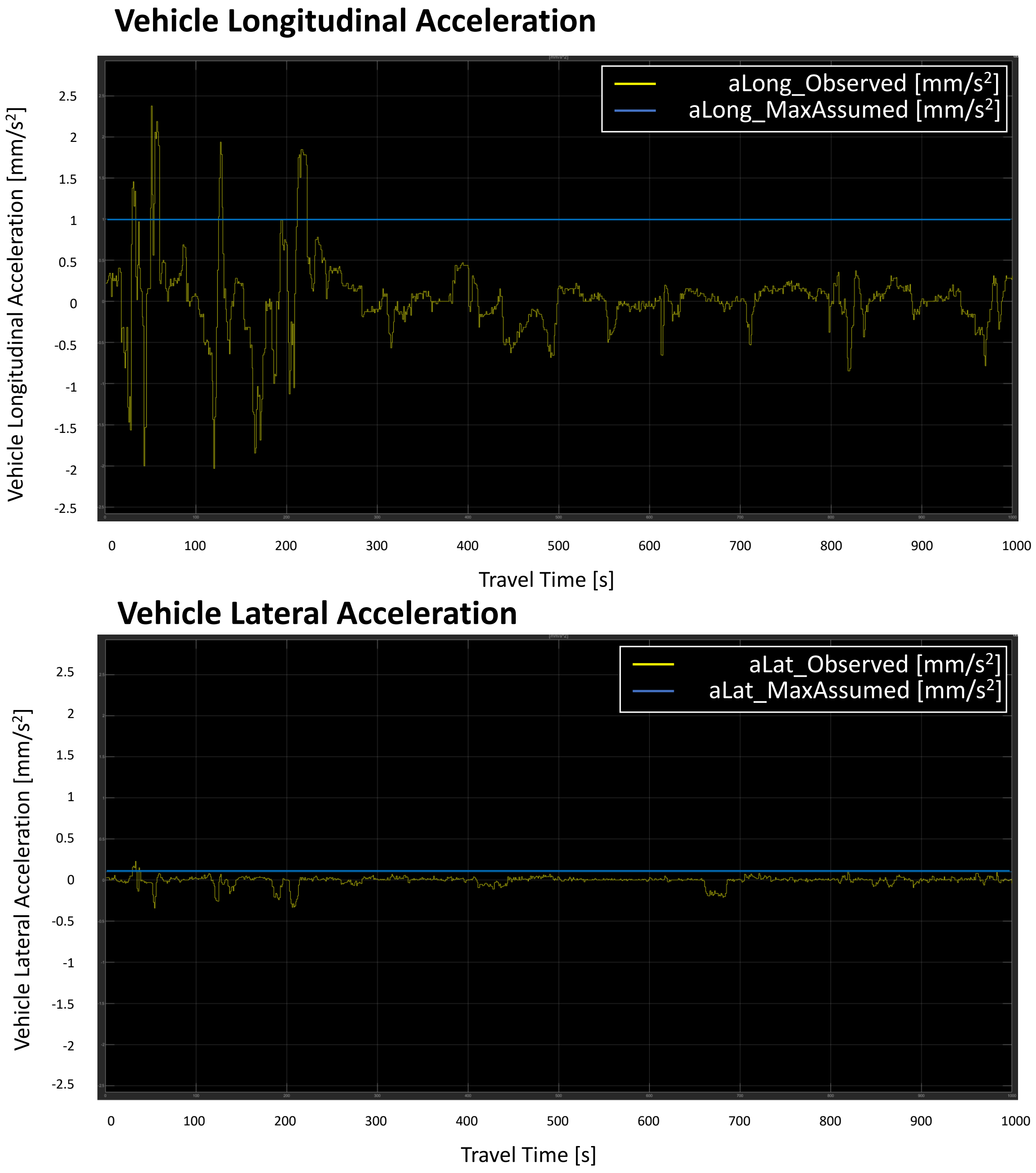


Figure A.3.: Vehicle Speed Estimation Function: Environment Assumptions Monitor for the Road Slope during the *Dynamic Driving* Scenario.

Notice that the environment assumptions regarding the road gradient and the lateral acceleration are valid, since the observed values distinctly lower than the maximum assumed value throughout the entire duration of the vehicle's trip.

With respect to the environment assumption monitor for the vehicle's longitudinal acceleration, the monitor trace displays data points situated above the line marking the maximum assumed value of the longitudinal acceleration for the first roughly 230 s of the vehicle's drive. Notice that during these approximately 230 s the curve of the longitudinal acceleration varies in the interval $[-2.0, 2.5]$ $\frac{mm}{s^2}$. This is indicative of abrupt acceleration and braking with rapid changes between the two types of actions. For this part of the vehicle's drive, the environment assumption is not valid, because the relative frequency of occurrence for small longitudinal acceleration values is less than the assumed probability of occurrence. After 230 s, the longitudinal acceleration stabilizes in the interval $[-0.8, 0.8]$ $\frac{mm}{s^2}$, which is situated below the maximum assumed value for the longitudinal acceleration of 1 $\frac{mm}{s^2}$ or 0.001 $\frac{m}{s^2}$. Thus, the environment assumption with respect to longitudinal acceleration becomes valid after approximately 230 s in the vehicle's trip.



Vehicle Longitudinal Acceleration

Vehicle Lateral Acceleration

Figure A.4.: Vehicle Speed Estimation Function: Environment Assumptions Monitors for the Longitudinal Acceleration and the Lateral Acceleration Environment during the *Dynamic Driving* Scenario.

Bibliography

[AAC$^+$05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. *ACM SIGPLAN Notices*, 40(10):345–364, 2005.

[AAHR16] Adina Aniculaesei, Daniel Arnsberger, Falk Howar, and Andreas Rausch. Towards the verification of safety-critical autonomous systems in dynamic environments. In *Proceedings of the The First Workshop on Verification and Validation of Cyber-Physical Systems, V2CPS@IFM 2016, Reykjavík, Iceland, June 4-5, 2016.*, pages 79–90, 2016.

[AAK$^+$22] Mohamed Toufik Ailane, Adina Aniculaesei, Christoph Knieke, Andreas Rausch, and Fauzi Scholichin. Towards specification completion for systems with emergent behavior based on devops, 2022. Accepted at 9th International Conference on Computational Science and Computational Intelligence (CSCI'22).

[ÁBD$^+$14] Erika Ábrahám, Bernd Becker, Christian Dehnert, Nils Jansen, Joost-Pieter Katoen, and Ralf Wimmer. Counterexample generation for discrete-time markov models: An introductory survey. In Marco Bernardo, editor, *Formal methods for executable software models*, volume 8483 of *LNCS sublibrary: SL 2 - Programming and software engineering*, pages 65–121. Springer, Cham, 2014.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Logic in computer science*, pages 414–425. IEEE Comput. Soc. Press, 1990.

[AD92] Rajeev Alur and David Dill. The theory of timed automata. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, pages 45–73, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[ADT14] Thomas Arts, Michele Dorigatti, and Stefano Tonetta. Making implicit safety requirements explicit: An autosar safety case. In *Computer Safety, Reliability, and Security. SAFECOMP 2014*, Lecture Notes in Computer

Science, pages 81–92. Springer Nature Switzerland AG, Cham, Switzerland, 2014.

[AFM16] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In Erika Ábrahám, Marcello Bonsangue, and Einar Broch Johnsen, editors, *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 47–64. Springer International Publishing, Cham, 2016.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AHDR18] Adina Aniculaesei, Falk Howar, Peer Denecke, and Andreas Rausch. Automated generation of requirements-based test cases for an adaptive cruise control system. In Cyrille Artho and Rudolf Ramler, editors, *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST@SANER)*, pages 11–15. IEEE, 2018.

[AL93] Mart\'ın Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[AL09] Husain Aljazzar and Stefan Leue. Generation of counterexamples for model checking of markov decision processes. In *QEST 2009*, pages 197–206, Los Alamitos Calif., 2009. IEEE Computer Society.

[Alu15] Rajeev Alur. *Principles of Cyber-Physical Systems.* MIT Press, Cambridge, MA, USA, 1 edition, April 2015.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[Ass17] Association of Transportation Safety Information Professionals. Ansi-d-16-2017: American national standard, 2017.

[ASSB96] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 269–276, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[AVR19a] Adina Aniculaesei, Andreas Vorwald, and Andreas Rausch. Automated generation of requirements-based test cases for an automotive function using the scade toolchain. In Nadia Abchiche-Mimouni, Sebastian Herold, Mirco Schindler, Christoph Knieke, Piotr Malak, and Tomasz Walkowiak, editors, *ADAPTIVE 2019: The Eleventh International Conference on Adaptive and Self-Adaptive Systems and Applications*, pages 69–74. IARIA, 2019.

[AVR19b] Adina Aniculaesei, Andreas Vorwald, and Andreas Rausch. Using the scade toolchain to generate requirements-based test cases for an adapative cruise control system. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 502–512. ACM/IEEE, 2019.

[AVZR21] Adina Aniculaesei, Andreas Vorwald, Meng Zhang, and Andreas Rausch. Architecture-based hybrid approach to verify safety-critical autonomotive system functions by combining data-driven and formal methods. In Cyrille Artho and Rudolf Ramler, editors, *2021 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 139–148. IEEE, 2021.

[AZR20] Adina Aniculaesei, Meng Zhang, and Andreas Rausch. Data-driven approach for accurate estimation and validation of the ego-vehicle speed. In Christoph Knieke, Mo Mansouri, and Giulio Telleschi, editors, *ICONS 2020: The Fifteenth International Conference on Systems*, pages 72–77. IARIA, 2020.

[BBLN17] Ezio Bartocci, Luca Bortolussi, Michele Loreti, and Laura Nenzi. Monitoring mobile and spatially distributed cyber-physical systems. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 146–155, New York, NY, USA, 09292017. ACM.

[BBM$^+$15] Ezio Bartocci, Luca Bortolussi, Dimitrios Milios, Laura Nenzi, and Guido Sanguinetti. Studying emergent behaviours in morphogenesis using signal spatio-temporal logic. In Alessandro Abate and David Šafránek, editors, *Hybrid systems biology*, volume 9271 of *Lecture notes in computer science, 0302-9743*, pages 156–172. Springer, Cham, 2015.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer eBook Collection. Springer, Berlin and Heidelberg, 2004.

[BCP07] Albert Benveniste, Benoit Caillaud, and Roberto Passerone. A generic model of contracts for embedded systems, 2007.

[BD97] Robert H. Bishop and Richard C. Modern control systems. th ed Dorf. *Modern control systems analysis and design using MATLAB and SIMULINK*. Addison-Wesley, Menlo Park, Calif and Harlow, 1997.

[BdA95] Andrea Bianco and Luca de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 499–513, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[BdAFK18] Christel Baier, Luca de Alfaro, Vojtěch Forejt, and Marta Kwiatkowska. Model checking probabilistic systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 963–999. Springer International Publishing, Cham, 2018.

[BDD⁺18] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. 10457:135–175, 2018.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[BDL12] Benedikt Bollig, Normann Decker, and Martin Leucker. Frequency linear-time temporal logic. In *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*, pages 85–92. IEEE, 2012.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[BFBBI] Marjory S. Blumenthal, Laura Fraade-Blanar, Ryan Best, and J. Luke Irwin. Safe enough: Approaches to assessing acceptable safety for automated vehicles.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification: Lectures on runtime verification - introductory and advanced topics. *LNCS*, 10457:1–33, 2018.

[BFL⁺18] Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, Joël Ouaknine, and James Worrell. Model checking real-time systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1001–1046. Springer International Publishing, Cham, 2018.

[BGP03] Howard Barringer, Dimitra Giannakopoulou, and Corina Păsăreanu. Proof rules for automated compositional verification. 09 2003.

[BH20] Simon Burton and Richard Hawkins. Assuring the safety of highly automated driving: state-of-the-art and research perspectives, 2020.

[BHKS12] T. Bures, P Hnetynka, P. Kroha, and V. Simko. Requirement specifications using natural languages. Technical Report D3S-TR-2012-05, Charles University, Dec 2012.

[BHL] Nikita Bhardwaj Haupt and Peter Liggesmeyer. Towards context-awareness for enhanced safety of autonomous vehicles.

[BHM+15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[Bit00] Friedemann Bitsch. Classification of safety requirements for formal verification of software models of industrial automation systems. In *13th International Conference on Software and Systems Engineering and their Applications (ICSSEA)*. CNAM – Paris, France, 2000.

[Bit01] Friedemann Bitsch. Safety patterns — the key to formal specification of safety requirements. In U. Voges, editor, *Computer safety, reliability and security*, volume 2187 of *Lecture notes in computer science, 0302-9743*, pages 176–189. Springer, Berlin and London, 2001.

[BJP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Saar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[BK98] Christel Baier and Marta Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT, Cambridge, Mass. and London, 2008.

[BKH99] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 146–161, Berlin, Heidelberg, 1999. Springer-Verlag.

[BKH+13] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The brics component model: A model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1758–1764, New York, NY, USA, 2013. Association for Computing Machinery.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4), 2011.

[BRB13] Ahmed Baig, Risza Rusli, and Azizul Buang. Reliability analysis using fault tree analysis: A review. *International Journal of Chemical Engineering and Applications*, 4:169–173, 01 2013.

[BRH10] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for runtime monitoring: from eagle to ruler. *Journal of Logic and Computation*, 20(3):675–706, 2010.

[BTFM08] Michael Bonner, Robert Taylor, Keith Fletcher, and Christopher Miller. Adaptive automation and decision aiding in the military fast jet domain. *Human Performance, Situation Awareness and Automation: User-Centred Design for the New Millennium*, 12 2008.

[BTM+18] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: verified controller executables from verified cyber-physical system models. *ACM SIGPLAN Notices*, 53(4):617–630, 2018.

[Buz19] Igor Buzhinsky. Formalization of natural language requirements into temporal logics: a survey. In *17th IEEE International Conference on Industrial Informatics (INDIN)*, pages 400–406. IEEE, 2019.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on concurreny and Petri nets*, volume 3098 of *Lecture notes in computer science, 0302-9743*, pages 87–124. Springer, Berlin and London, 2004.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing.

[CDG07] Jose R. Celaya, Alan A. Desrochers, and Robert J. Graves. Modeling and analysis of multi-agent systems using petri nets. In *IEEE International Conference on Systems, Man and Cybernetics, 2007*, pages 1439–1444, Piscataway, NJ, 2007. IEEE Service Center.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, Berlin, Heidelberg, 1981. Springer-Verlag.

[CFAI17] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In Adrian Francalanza and Gordon J. Pace, editors, *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017*, volume 254 of *EPTCS*, pages 15–28, 2017.

[CFK+13] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. Prism-games: A model checker for stochastic multi-player games.

In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 185–191, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[CFL$^+$20] Rafael C. Cardoso, Marie Farrell, Matt Luckcuck, Angelo Ferrando, and Michael Fisher. Heterogeneous verification of an autonomous curiosity rover. In Ritchie Lee, Susmit Jha, and Anastasia Mavridou, editors, *NASA formal methods*, volume 12229 of *LNCS Sublibrary: SL2 - Programming and software engineering*, pages 353–360. Springer, Cham, Switzerland, 2020.

[CG04] Frank Ciesinski and Marcus Größer. On probabilistic computation tree logic. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems: A Guide to Current Research*, pages 147–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[CHMS06] J. Connelly, W. S. Hong, R. B. Mahoney, Jr., and D. A. Sparrow. Current challenges in autonomous vehicle development. In Grant R. Gerhart, Charles M. Shoemaker, and Douglas W. Gage, editors, *Unmanned Systems Technology VIII*, volume 6230, pages 115–125. SPIE, 2006.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[CIT20] D. M. Chack, S. G. Inks, and C. M. Thompson. Vehicle speed control system: Utility, 2020.

[CK95] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. *SIGSOFT Softw. Eng. Notes*, 20(4):140–150, 1995.

[CK96] Shing-Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5:334–377, 10 1996.

[CKSW13] Taolue Chen, Marta Kwiatkowska, Aistis Simaitis, and Clemens Wiltsche. Synthesis for multi-objective stochastic games: An application to autonomous urban driving. In Kaustubh Joshi, editor, *Proc. 10th International Conference on Quantitative Evaluation of SysTems (QEST'13)*, volume 8054 of *LNCS Sublibrary: SL 1 - Theoretical computer science and general issues*, pages 322–337. Springer, Heidelberg, 2013.

[CL07] Hugo Costelha and Pedro Lima. Modelling, analysis and execution of robotic tasks using petri nets. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2007*, pages 1449–1454, Piscataway, NJ, 2007. IEEE Service Center.

[CPP17] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *TASE 2017 - 11th International Symposium on Theoretical Aspects of Software Engineering*, pages 1–10, Nice, France, September 2017.

[CPS+18] Ian Colwell, Buu Phan, Shahwar Saleem, Rick Salay, and Krzysztof Czarnecki. An automated vehicle safety concept based on runtime restriction of the operational design domain. In *Intelligent Vehicles Symposium*, pages 1910–1917. 2018.

[CTT19a] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Assumption-based runtime verification with partial observability and resets. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification*, pages 165–184, Cham, 2019. Springer International Publishing.

[CTT19b] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. Nurv: A nuxmv extension for runtime verification. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification*, pages 382–392, Cham, 2019. Springer International Publishing.

[CZ97] Antonio Cau and Hussein Zedan. Refining interval temporal logic specifications. In Miquel Bertran and Teodor Rus, editors, *Transformation-based reactive systems development*, volume 1231 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 7–15, New York, NY, USA, 1998. Association for Computing Machinery.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. Association for Computing Machinery.

[dAFH+04] Luca de Alfaro, Marco Faella, Thomas A. Henzinger, Rupak Majumdar, and Mariëlle Stoelinga. Model checking discounted temporal properties. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 77–92, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5), 2001.

[DAR+15] P. Damanab, Seyed Shamseddin Alizadeh, Yahya Rasoulzadeh, P. Moshashaie, and Sakineh Varmazyar. Failure modes and effects analysis (fmea) technique: a literature review. 4:1–6, 01 2015.

[DC08] Benoît Delahaye and Benoit Caillaud. A model for probabilistic reasoning on assume/guarantee contracts, 2008.

[DCL11] Benoît Delahaye, Benoît Caillaud, and Axel Legay. Probabilistic contracts: A compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. *Form. Methods Syst. Des.*, 38(1):1–32, feb 2011.

[DDS17] Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. Combining model checking and runtime verification for safe robotics. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 172–189, Cham, 2017. Springer International Publishing.

[Del10] Benoît Delahaye. *Modular Specification and Compositional Analysis of Stochastic Systems*. Phd, Université Rennes 1, 2010.

[DF20] Louise A. Dennis and Michael Fisher. Verifiable self-aware agent-based autonomous systems. *Proceedings of the IEEE*, 108(7):1011–1026, 2020.

[DFL+16] Louise A. Dennis, Michael Fisher, Nicholas K. Lincoln, Alexei Lisitsa, and Sandor M. Veres. Practical verification of decision-making in agent-based autonomous systems. *Automated Software Engineering*, 23(3):305–359, 2016.

[DFSW16] Louise Dennis, Michael Fisher, Marija Slavkovik, and Matt Webster. Formal verification of ethical choices in autonomous systems. *Robotics and Autonomous Systems*, 77:1–14, 2016.

[DFVA09] Jordi Dunjó, Vasilis Fthenakis, Juan Vilchez, and Josep Arnaldos. Hazard and operability (hazop) analysis. a literature review. *Journal of hazardous materials*, 173:19–32, 09 2009.

[DFWB12] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[DGRW04] H. Dittrich, V. Gärtner, R. Rinck, and V. Wehren. Method for determining a vehicle reference speed: Utility, 2004.

[DH20a] Hoang Tung Dinh and Tom Holvoet. Verifying autonomous decision making against environment assumptions: An experience report. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 327–335. IEEE, 11/9/2020 - 11/11/2020.

[DH20b] Hoang Tung Dinh and Tom Holvoet. A framework for verifying autonomous robotic agents against environment assumptions. 10 2020.

[Dij72] E. W. Dijkstra. The humble programmer. *Communications of ACM*, 15(10):859–866, 1972.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker, 2017.

[DL96] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 179–190. ACM Press, 1996.

[DM08] Louise Dennis and Berndt Müller. Gwendolen: A bdi language for verifiable agents. 01 2008.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Don10] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer aided verification*, volume 6174 of *LNCS sublibrary. SL 1, Theoretical computer science and general issues*, pages 167–170. Springer, Berlin, 2010.

[dpa22] dpa. Autonomes fahren: Mercedes startet verkauf von „drive pilot“. *t3n digital pioneers*, 2022.

[DWFZ12] Clare Dixon, Alan F.T. Winfield, Michael Fisher, and Chengxiu Zeng. Towards temporal verification of swarm robotic systems. *Robotics and Autonomous Systems*, 60(11):1429–1441, 2012.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[EKVY07] K. Etessami, M. Kwiatkowska, M. Y. Vardi, and M. Yannakakis. Multi-objective model checking of markov decision processes. In *TACAS*, pages 50–65. 2007.

[Eura] Euro NCAP. Assessment protocol - safety assist.

[Eurb] European Economic Community. Council directive of 26 june 1975 on the approximation of the laws of the member states relating to the reverse and speedometer equipment of motor vehicles: 75/443/eec.

[FBC⁺20] Simon Foster, James Baxter, Ana Cavalcanti, Jim Woodcock, and Frank Zeyda. Unifying semantic foundations for automated verification tools in isabelle/utp. *Science of Computer Programming*, 197:102510, 2020.

[FCD⁺19] M. Farrell, R. C. Cardoso, L. Dennis, C. Dixon, Michael Fisher, G. Kourtis, A. Lisitsa, Matt Luckcuck, and M. Webster. Modular verification of autonomous space robotics. *Arxiv*, abs/1908.10738, 2019.

[FCF⁺21] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini, and Viviana Mascardi. Rosmonitoring: A runtime verification framework for ros. In Abdelkhalick Mohammad, Xin Dong, and Matteo Russo, editors, *Towards autonomous robotic systems*, volume 12228 of *LNCS sublibrary: SL7 - Artificial intelligence*, pages 387–399. Springer, Cham, Switzerland, 2021.

[FDA⁺18a] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. Recognising assumption violations in autonomous systems verification. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, pages 1933–1935, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[FDA⁺18b] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. Verifying and validating autonomous systems: Towards an integrated approach. In Christian Colombo and Martin Leucker, editors, *Runtime verification*, volume 11237 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 263–281. Springer, Cham, Switzerland, 2018.

[FDW13] Michael Fisher, Louise Dennis, and Matt Webster. Verifying autonomous systems. *Commun. ACM*, 56(9):84–93, September 2013.

[Fen14] Lu Feng. *On learning assumptions for compositional verification of probabilistic systems*. Phd, University of Oxford, UK, 2014.

[Fer19] Angelo Ferrando. The early bird catches the worm: First verify, then monitor! *Science of Computer Programming*, 172:160–179, 2019.

[FGR$^+$94] A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Form. Methods Syst. Des.*, 4(3):243–263, May 1994.

[FHKP11] Lu Feng, Tingting Han, Marta Kwiatkowska, and David Parker. Learning-based compositional verification for synchronous probabilistic systems. In *9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, volume 6996 of *LNCS*, pages 511–521, Taipei, Taiwan, 2011. Springer.

[FJN$^+$11] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software engineering and formal methods*, volume 7041 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 204–220. Springer, Heidelberg, 2011.

[FKN$^+$11] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Quantitative multi-objective verification for probabilistic systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 112–127, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[FKNP11] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In Marco Bernardo and Valérie Issarny, editors, *Formal methods for eternal networked software systems*, volume 6659 of *Lecture notes in computer science, 0302-9743*, pages 53–113. Springer, Heidelberg, 2011.

[FKP10] Lu Feng, Marta Kwiatkowska, and David Parker. Compositional verification of probabilistic systems using learning. In *2010 Seventh International Conference on the Quantitative Evaluation of Systems*, pages 133–142. IEEE, 9/15/2010 - 9/18/2010.

[FMR00] S. Flake, W. Müller, and J. Ruf. Structured english for model checking specification. In *MBMV*, 2000.

[FMR$^+$21] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd-Holger Schlingloff, Michael Winikoff, and Neil Yorke-Smith. Towards a framework for certification of reliable autonomous systems. *Auton. Agents Multi Agent Syst.*, 35:8, 2021.

[FPEN15] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback control of dynamic systems.* Pearson, Boston, seventh edition edition, 2015.

[Fra20] Luca Franceschini. *RML: Runtime Monitoring Language.* Phd, Universita di Genova, Italy, 2020.

[Gam11] Erich Gamma. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley professional computing series. Addison-Wesley, Boston, 39. printing edition, 2011.

[GBPG08] Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In Aarti Gupta and Sharad Malik, editors, *Computer aided verification*, Lecture notes in computer science, 0302-9743, pages 135–148. Springer, Berlin, 2008.

[GFN19] Mario Gleirscher, Simon Foster, and Yakoub Nemouchi. Evolution of formal model-based assurance cases for autonomous robots. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 87–104, Cham, 2019. Springer International Publishing.

[GGP07] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–307, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[GJD13] Meng Guo, Karl H. Johansson, and Dimos V. Dimarogonas. Revising motion planning under linear temporal logic specifications in partially known workspaces. In *2013 IEEE International Conference on Robotics and Automation*, pages 5025–5032, 2013.

[GK19] Mario Gleirscher and Stefan Kugele. Assurance of system safety: A survey of design and argument patterns. *Arxiv*, 1902.05537, 2019.

[Gle14] Mario Gleirscher. *Behavioural Safety of Technical Systems.* Phd, TU München, Munich, Germany, 2014.

[GNP18] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Păsăreanu. Compositional reasoning. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 345–383. Springer International Publishing, Cham, 2018.

[GPB05] Dimitra Giannakopoulou, Corina S. Păsăreanu, and Howard Barringer. Component verification with automatically generated assumptions. *Automated Software Engineering*, 12(3):297–320, 2005.

[Gre11] Joel Greenyer. *Scenario-based Design of Mechatronic Systems.* Phd, Universität Paderborn, Germany, 2011.

[GRR$^+$16] Tobias Gindele, Bernhard Rumpe, Andreas Rausch, Martin Vossiek, Peter Gulden, Christian Schlegel, and Christian Verbeek. iserveU: Intelligente modulare Serviceroboter-Funktionalitäten im menschlichen Umfeld am Beispiel von Krankenhäusern, 2016. Final report, Funding code 01IM12008A, unpublished.

[Gru08] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. Association for Computing Machinery.

[HABR22] Felix Helsch, Iqra Aslam, Abhishek Buragohain, and Andreas Rausch. Qualitative monitors based on the connected dependability cage approach. In *The 14th International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2022)*, pages 46–55. IARIA, 2022.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hav15] Klaus Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.

[Hen18] Christian Hensel. *The Probabilistic Model Checker Storm: Symbolic Methods for Probabilistic Model Checking.* Phd, RWTH Aachen, Germany, 2018.

[HEZ$^+$14] Jeff Huang, Cansu Erdogan, Yi Zhang, Brandon Moore, Qingzhou Luo, Aravind Sundaresan, and Grigore Rosu. Rosrv: Runtime verification for robots. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime verification*, volume 8734 of *LNCS sublibrary: SL 2 - Programming and software engineering*, pages 247–254. Springer, Cham, 2014.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HJK$^+$15] Iman Haghighi, Austin Jones, Zhaodan Kong, Ezio Bartocci, Radu Gros, and Calin Belta. Spatel - a novel spatial-temporal logic and its applications to networked systems. In Antoine Girard and Sriram Sankaranarayanan, editors, *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 189–198, New York, NY, USA, 04142015. ACM.

[HK07] Tingting Han and Joost-Pieter Katoen. Counterexamples in probabilistic model checking. In Orna Grumberg and Michael Huth, editors, *Tools and algorithms for the construction and analysis of systems*, volume 4424 of *Lecture notes in computer science, 0302-9743*, pages 72–86. Springer, Berlin and London, 2007.

[HKNP06] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and algorithms for the* *construction and analysis of systems*, volume 3920 of *Lecture notes in computer science, 0302-9743*, pages 441–444. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[HKW$^+$08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. Cocome - the common component modeling example. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *The Common Component Modeling Example: Comparing Software Component Models*, pages 16–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[HL19] Nikita Bhardwaj Haupt and Peter Liggesmeyer. A runtime safety monitoring approach for adaptable autonomous systems. In Alexander Romanovsky, Elena Troubitsyna, Ilir Gashi, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer safety, reliability, and security*, volume 11699 of *LNCS sublibrary. SL2 - Programming and software engineering*, pages 166–177. Springer, Cham, 2019.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Softw. Syst. Model.*, 7(2):237–252, 2008.

[HMA03] Hui-Min Huang, Elena Messina, and James Albus. Autonomy level specification for intelligent autonomous vehicles: Interim progress report. In *Proceedings of 2003 Performance Metrics for Intelligent Systems (PerMIS) Workshop*, 2003.

[HMU14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Pearson custom library. Pearson Education, Harlow, Essex, pearson new international ed. edition, 2014.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol02] Gerard J. Holzmann. The logic of bugs. *SIGSOFT Softw. Eng. Notes*, 27(6):81–87, 2002.

[How60] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press and John Wiley & Sons Inc., MA, USA, 1960.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, pages 477–498. Springer, Berlin, Heidelberg, 1985.

[HR17] Klaus Havelund and Giles Reger. Runtime verification logics a language design perspective. In K. G. Larsen, Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare, editors, *Models, algorithms, logics and tools*, volume 10460 of *Lecture notes in computer science, 0302-9743*, pages 310–338. Springer, Cham, 2017.

[Int97a] International Electrotechnical Commission. Iec 61508-1:1997 - functional safety of electrical/electronic/programmable electronic safety related systems: Part 1: General requirements, 1997.

[Int97b] International Electrotechnical Commission. Iec 61508-4:1997 - functional safety of electrical/electronic/programmable electronic safety related systems: Part 4: Definitions and abbreviations, 1997.

[Int11a] International Organization for Standardization. Iso 26262-10:2011: Road vehicles - functional safety: Part 10: Guideline to iso 26262, 2011.

[Int11b] International Organization for Standardization. Iso 26262-1:2011: Road vehicles - functional safety: Part 1: Vocabulary, 2011.

[Int11c] International Organization for Standardization. Iso 26262-3:2011: Road vehicles - functional safety: Part 3: Concept phase, 2011.

[Int11d] International Organization for Standardization. Iso 26262-4:2011: Road vehicles - functional safety: Part 4: Product development at the system level, 2011.

[Int11e] International Organization for Standardization. Iso 26262-5:2011: Road vehicles - functional safety: Part 5: Product development at the hardware level, 2011.

[Int11f] International Organization for Standardization. Iso 26262-6:2011: Road vehicles - functional safety: Part 6: Product development at the software level, 2011.

[Int11g] International Organization for Standardization. Iso 26262-9:2011: Road vehicles - functional safety: Part 9: Automotive safety integrity level (asil)-oriented and safety-oriented analyses, 2011.

[Int19] International Organization for Standardization. Iso/pas 21448 - road vehicles - safety of the intended functionality, 2019.

[IQV16] Paolo Izzo, Hongyang Qu, and Sandor M. Veres. A stochastically verifiable autonomous control architecture with reasoning. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 2016.

[ISO10] ISO, IEC, IEEE. Iso/iec/ieee 24765:2010 - systems and software engineering: Vocabulary, 2010.

[Jan15] Nils Jansen. *Counterexamples in Probabilistic Verification.* Phd, RWTH Aachen, Germany, 2015.

[JG16] Bertrand Jeannet and Fabien Gaucher. Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016.

[JO01] J. Bezivin and O. Gerbe. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280, 2001.

[Jon83a] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[Jon83b] Cliff Jones. Specification and design of (parallel) programs. volume 83, pages 321–332, 01 1983.

[JPQ$^+$16] Rainer Joppich, Andre Pflüger, Stefan Queins, Chris Rupp, Kristina Schöne, Achim Stuy, and Alexander Vöge. Master - schablone für alle fälle. https://www.sophist.de/MASTeR-Broschuere, 2016.

[Kaf12] Peter Kafka. The automotive standard iso 26262, the innovative driver for enhanced safety assessment & technology for motor cars. In Feng Changgen and Li Shengcai, editors, *2012 International Symposium on Safety Science and Technology*, volume 45 of *Procedia Engineering*, pages 2–10. Elsevier Ltd., 2012.

[Kat16] Joost-Pieter Katoen. The probabilistic model checking landscape*. In *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15, 2016.

[KC05] S. Konrad and B.H.C. Cheng. Real-time specification patterns. In *ICSE'05*, pages 372–381. IEEE, 2005.

[KDF12] Savas Konur, Clare Dixon, and Michael Fisher. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, 60(2):199–213, 2012.

[KDJ$^+$16] James Kapinski, Jyotirmoy V. Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts. Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems*, 36(6):45–64, 2016.

[KFK14] Aaron Kane, Thomas Fuhrman, and Philip Koopman. Monitor based oracles for cyber-physical system testing: Practical experience report. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 148–155. IEEE, 6/23/2014 - 6/26/2014.

[KGFP08] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, (22):1343–1359, 2008.

[KJ10] Zvi Kohavi and Niraj K. Jha. *Switching and finite automata theory.* Cambridge University Press, Cambridge UK and New York, 3rd ed. edition, 2010.

[KM16] Felix Kossak and Atif Mashkoor. How to select the suitable formal method for an industrial application: A survey, 2016.

[Kni02] John C. Knight. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In Anthony J. Field, editor, *Computer performance evaluation*, volume 2324 of *Lecture notes in computer science, 0302-9743*, pages 200–204. Springer, Berlin and London, 2002.

[KNPQ10] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-guarantee verification for probabilistic systems. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 23–37, Berlin, Heidelberg, 2010. Springer-Verlag.

[KNPQ13] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Compositional probabilistic verification through multi-objective model checking. *Information and Computation*, 232:38–65, 2013.

[KO19] Philip Koopman and Beth Osyk. Safety argument considerations for public road testing of autonomous vehicles. In *SAE Technical Paper Series*, SAE Technical Paper Series. SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2019.

[Kos94] F. Kost. Speed estimation process: Utility, 1994.

[KPC12] Anvesh Komuravelli, Corina S. Pasareanu, and Edmund M. Clarke. Assume-guarantee abstraction refinement for probabilistic systems. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 310–326, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Kri20] Thomas Krismayer. *Automatic Mining of Constraints for Eventbased Systems Monitoring.* Phd, Johannes Kepler University Linz, Austria, 2020.

[KSZ14] Joost-Pieter Katoen, Lei Song, and Lijun Zhang. Probably safe or live. In Thomas Henzinger and Dale Miller, editors, *Proceedings of the Joint Meeting* *of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACMIEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, New York, NY, 2014. ACM.

[KW16] Phillip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. In *SAE World Congress*. 2016.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lam01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, page 249, USA, 2001. IEEE Computer Society.

[Lan08] Arnaud Lanoix. Event-b specification of a situated multi-agent system: Study of a platoon of vehicles. In *Proceedings*, pages 297–304, Las Alamitos Calif., 2008. IEEE Computer Society.

[LBBZ97] U. Lefarth, U. Baum, T. Beck, and T. Zurawka. Ascet-sd - development environment for embedded control systems. *IFAC Proceedings Volumes*, 30(4):85–90, 1997.

[LDS11] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *Proceedings of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*, MEMOCODE '11, page 43–50, USA, 2011. IEEE Computer Society.

[LDSW09] Hui Liang, Jin Song Dong, Jing Sun, and W. Eric Wong. Software monitoring through formal specification animation. *Innovations in Systems and Software Engineering*, 5(4):231–241, 2009.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for programming, artificial intelligence, and reasoning*, volume 6355 of *LNCS sublibrary. SL 7, Artificial intelligence*, pages 348–370. Springer, Berlin, 2010.

[LFD$^+$19] Matt Luckcuck, Marie Farell, Louise Abigail Dennis, Claire Dixon, and Michael Fisher. A summary of formal specification and verification of autonomous robotic systems. In *IFM*, 2019.

[Li14] Wenchao Li. *Specification Mining: New Formalisms, Algorithms and Applications.* Phd, University of California, Berkely, USA, 2014.

[Lig09] P. Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software.* Springer Spektrum, Heidelberg, Germany, 2009.

[LNSV+17] Jiwei Li, Pierluigi Nuzzo, Alberto Sangiovanni-Vincentelli, Yugeng Xi, and Dewei Li. Stochastic contracts for cyber-physical system design under probabilistic requirements. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '17, pages 5–14, New York, NY, USA, 2017. Association for Computing Machinery.

[LPN11] Sarah M. Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In Michael Butler and Wolfram Schulte, editors, *FM 2011*, volume 6664 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 42–56. Springer, Heidelberg, 2011.

[LS07] Martin Leucker and César Sánchez. Regular linear temporal logic. In C. B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Theoretical aspects of computing - ICTAC 2007*, volume 4711 of *Lecture notes in computer science, 0302-9743*, pages 291–305. Springer, Berlin, 2007.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[LSSS14] Wenchao Li, Dorsa Sadigh, S. Shankar Sastry, and Sanjit A. Seshia. Synthesis for human-in-the-loop control systems. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–484, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[Luc21] Matt Luckcuck. Using formal methods for autonomous systems: Five recipes for formal verification. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, page 1748006X2110349, 2021.

[LYX19] Peng Liu, Run Yang, and Zhigang Xu. How safe is safe enough for self-driving vehicles? *Risk analysis : an official publication of the Society for Risk Analysis*, 39(2):315–325, 2019.

[Mau19] Malte Mauritz. *Engineering of safe autonomous vehicles through seamless integration of system development and system operation.* Verlag Dr. Hut, München, 2019.

[MBL+13] Mieke Massink, Manuele Brambilla, Diego Latella, Marco Dorigo, and Mauro Birattari. On the use of bio-pepa for modelling and analysing collective behaviours in swarm robotics. *Swarm Intelligence*, 7(2):201–228, 2013.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[MFG$^+$22] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[MGVP17] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher, and André Platzer. Formal verification of obstacle avoidance and navigation of ground robots. *The International Journal of Robotics Research*, 36(12):1312–1340, 2017.

[MHB13] Atif Mashkoor, Osman Hasan, and Wolfgang Beer. Using probabilistic analysis for the certification of machine control systems. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar R. Weippl, and Li D. Xu, editors, *Security engineering and intelligence informatics*, volume 8128 of *LNCS sublibrary. SL 3, Information systems and application, incl. Internet/Web and HCI*, pages 305–320. Springer, Heidelberg, 2013.

[MJ11] Atif Mashkoor and Jean-Pierre Jacquot. Utilizing event-b for domain engineering: a critical analysis. *Requirements Engineering*, 16(3):191–207, 2011.

[MJS] Atif Mashkoor, Jean-Pierre Jacquot, and Jeanine Souquières. Transformation heuristics for formal requirements validation by animation. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert)*.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166. Springer, Berlin, Heidelberg, 2004.

[Mor21] James Morris. Why is tesla's full self-driving only level 2 autonomous? *Forbes*, 2021.

[MP16] Stefan Mitsch and André Platzer. Modelplex: verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.*, 49(1-2):33–74, 2016.

[MP18] Stefan Mitsch and André Platzer. Verified runtime validation for partially observable hybrid systems. 2018.

[MR] Shahar Maoz and Jan Oliver Ringert. Spectra: a specification language for reactive systems. *Software and Systems Modeling*.

[MR15] Shahar Maoz and Jan Oliver Ringert. Gr(1) synthesis for ltl specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 96–106, New York, NY, USA, 2015. Association for Computing Machinery.

[MR16] Shahar Maoz and Jan Ringert. Synthesizing a lego forklift controller in gr(1): A case study. *Electronic Proceedings in Theoretical Computer Science*, 202:58–72, 02 2016.

[MRW06] T. Merz, P. Rudol, and M. Wzorek. Control system framework for autonomous robots based on extended state machines. In *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, page 14. IEEE, 19-21 July 2006.

[MS12] Shahar Maoz and Yaniv Saar. Assume-guarantee scenarios: Semantics and synthesis. In Robert France, editor, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 335–351. Springer, Heidelberg, 2012.

[MTBP19] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. Psalm: Specification of dependable robotic missions. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*, pages 99–102. IEEE Press, 2019.

[MTP$^+$19] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. Specification patterns for robotic missions. *Arxiv*, abs/1901.02077, 2019.

[MV09] L. Molnar and S. M. Veres. System verification of autonomous underwater vehicles by model checking. In *OCEANS 2009-EUROPE*, pages 1–10, 2009.

[NB09] Allen P. Nikora and Galen Balcom. Automated identification of ltl patterns in natural language requirements. In *Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, ISSRE'09, pages 185–194. IEEE Press, 2009.

[NBC$^+$15] Laura Nenzi, Luca Bortolussi, Vincenzo Ciancia, Michele Loreti, and Mieke Massink. Qualitative and quantitative monitoring of spatio-temporal properties. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime verification*, volume 9333 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 21–37. Springer, Cham, 2015.

[Nee19] Michael A. Nees. Safer than the average human driver (who is less safe than me)? examining a popular safety benchmark for self-driving cars. *Journal of Safety Research*, 69:61–68, 2019.

[NF96] Rani Nelken and Nissim Francez. Automatic translation of natural language system specifications into temporal logic. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 360–371, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[NFGK] Yakoub Nemouchi, Simon Foster, Mario GLEIRSCHER, and Tim Kelly. Mechanised assurance cases with integrated formal methods in isabelle.

[Nik05] Allen P. Nikora. Classifying requirements: towards a more rigorous analysis of natural-language specifications. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 10 p–300, 2005.

[Nip02] Tobias Nipkow, editor. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Springer eBook Collection Computer Science*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2002.

[NLSV$^+$19] Pierluigi Nuzzo, Jiwei Li, Alberto L. Sangiovanni-Vincentelli, Yugeng Xi, and Dewei Li. Stochastic assume-guarantee contracts for cyber-physical system design. *ACM Trans. Embed. Comput. Syst.*, 18(1), 2019.

[OAH$^+$14] Matthew O'Brien, Ronald C. Arkin, Dagan Harrington, Damian Lyons, and Shu Jiang. Automatic verification of autonomous robot missions. In Davide Brugali, editor, *Simulation, modeling, and programming for autonomous robots*, volume 8810 of *LNCS sublibrary. SL 7, Artificial intelligence*, pages 462–473. Springer, Cham, 2014.

[Obj05] Object Management Group. Uml profile for schedulability, performance, and time specification, 2005.

[OG76] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[PCR$^+$13] Edson Prestes, Joel Luis Carbonera, Sandro Rama Fiorini, Vitor A. M. Jorge, Mara Abel, Raj Madhavan, Angela Locoro, Paulo Goncalves, Marcos E. Barreto, Maki Habib, Abdelghani Chibani, Sébastien Gérard, Yacine Amirat, and Craig Schlenoff. Towards a core ontology for robotics and automation. *Robotics and Autonomous Systems*, 61(11):1193–1204, 2013.

[Pel01] Doron A. Peled. *Software Reliability Methods*. Texts in Computer Science. Springer, New York, NY, 2001.

[Pet20] Dieter Petereit. Level-3-automation: Mercedes-benz drive pilot fährt bis 60 stundenkilometer allein. *t3n digital pioneers*, 2020.

[PG06] Corina S. Păsăreanu and Dimitra Giannakopoulou. Towards a compositional spin. In Antti Valmari, editor, *Model checking software*, volume 3925 of *Lecture notes in computer science, 0302-9743*, pages 234–251. Springer, Berlin, 2006.

[PGGB+08] Corina S. Păsăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Form. Methods Syst. Des.*, 32(3):175–205, 2008.

[Pla08] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[PMM+07] C. Ponsard, P. Massonet, J. F. Molderez, A. Rifaut, A. van Lamsweerde, and H. Tran Van. Early verification and validation of mission critical systems. *Form. Methods Syst. Des.*, 30(3):233–247, 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 10/31/1977 - 11/2/1977.

[Pnu85] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Saar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, model checking, and abstract interpretation*, volume 3855 of *Lecture notes in computer science, 0302-9743*, pages 364–380. Springer, Berlin and London, 2006.

[Put94] Martin L. Puterman. *Markov Decision Processes.* John Wiley & Sons, Inc, Hoboken, NJ, USA, 1994.

[RAO92] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *International Conference on Software Engineering*, pages 105–118. IEEE, 1992.

[Rao96] Anand S. Rao. Decision procedures for prepositional linear-time belief-desire-intention logics. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *Intelligent Agents II Agent Theories, Architectures, and Languages*, pages 33–48, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[RB08] Andreas Rausch and Manfred Broy. Die v-modell xt grundlagen. In Reinhard Höhn and Stephan Höppner, editors, *Das V-Modell XT*, eXamen.press. Springer-Verlag, 2008.

[RCSL21] Ivan Ruchkin, Matthew Cleaveland, Oleg Sokolsky, and Insup Lee. Confidence monitoring and composition for dynamic assurance of learning-enabled autonomous systems. In Ezio Bartocci, Yliès Falcone, and Martin Leucker, editors, *Formal Methods in Outer Space*, volume 13065 of *Lecture Notes in*

Computer Science, pages 137–146. Springer International Publishing, Cham, 2021.

[Rod15] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[RP13] Andy Ruina and Rudra Pratap. *Introduction to Statics and Dynamics*. Oxford University Press, 2013.

[RRRW14] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *ArXiv*, abs/1505.00904, 2014.

[RS93] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[RS14] Chris Rupp and Die SOPHISTen. *Requirements Engineering and Management*, chapter 6, pages 115–156. Carl Hanser Verlag, Munich, Germany, 6 edition, 2 2014.

[RSS$^+$20] Jan Reich, Daniel Schneider, Ioannis Sorokos, Yiannis Papadopoulos, Tim Kelly, Ran Wei, Eric Armengaud, and Cem Kaypmaz. Engineering of runtime safety monitors for cyber-physical systems with digital dependability identities. In Antonio Casimiro, Frank Ortmeier, Friedemann Bitsch, and Pedro M. Ferreira, editors, *Computer safety, reliability, and security*, volume 12234 of *LNCS Sublibrary: SL2 - Programming and software engineering*, pages 3–17. Springer, Cham, Switzerland, 2020.

[RT20] Jan Reich and Mario Trapp. Sinadra: Towards a framework for assurable situation-aware dynamic risk assessment of autonomous vehicles. In *16th European Dependable Computing Conference*, pages 47–50, Los Alamitos, CA, 2020. IEEE Computer Society, Conference Publishing Services.

[RTC11] RTCA, Inc. Rtca 178c - software considerations in airborne systems and equipment certification, 2011.

[SAE18] SAE International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, 2018.

[Sch21] Raimund Schesswendter. Autonomes fahren: Das bedeuten level 0 bis 5. *t3n digital pioneers*, 2021.

[SdV04] Ana Sokolova and Erik P. de Vink. Probabilistic automata: System types, parallel composition and comparison. In Christel Baier, editor, *Validation of stochastic systems*, volume 2925 of *Lecture notes in computer science, Tutorial 0302-9743*, pages 1–43. Springer, Berlin and London, 2004.

[Seg95] Roberto Segala. *Modeling and Verification of Randomized Distributed Real -Time Systems*. Phd, Massachusetts Institute of Technology, Massachusetts, USA, 1995.

[Ses19] Sanjit A. Seshia. Introspective environment modeling. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime verification*, volume 11757 of *LNCS Sublibrary: SL2 - Programming and software engineering*, pages 15–26. Springer, Cham, Switzerland, 2019.

[SHL17] D.P. Schmitt, A. Hasegawa, and S. Lachmayr. Systems and methods for determining a speed limit violation: Utility, 2017.

[SKA13] Stefan Mitsch, Khalil Ghorbal, and André Platzer. On provably safe obstacle avoidance for autonomous robotic ground vehicles. In *Robotics: Science and Systems*, 2013.

[SLS14a] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations*, chapter 2, pages 5–38. Rocky Nook Inc., Santa Barbara, CA, USA, 4 edition, 2 2014.

[SLS14b] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations*, chapter 4, pages 79–104. Rocky Nook Inc., Santa Barbara, CA, USA, 4 edition, 2 2014.

[SLS14c] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations*, chapter 3, pages 39–78. Rocky Nook Inc., Santa Barbara, CA, USA, 4 edition, 2 2014.

[Som14a] Ian Sommerville. *Software Engineering*, chapter 2, pages 29–57. Addison-Wesley, Boston, MA, USA, 10 edition, 9 2014.

[Som14b] Ian Sommerville. *Software Engineering*, chapter 5, pages 82–117. Addison-Wesley, Boston, MA, USA, 10 edition, 9 2014.

[SS71] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. *Proceedings of the Symposium on Computers and Automata*, 21, 01 1971.

[SS11] David J. Smith and Kenneth G. L. Simpson. *The Safety Critical Systems Handbook: A Straightforward Guide to Functional Safety: IEC 61508 (2010 Edition) and Related Standards*, chapter 1, pages 3–20. Elsevier, Oxford, UK, 3 edition, 2011.

[SS16] Sanjit A. Seshia and Dorsa Sadigh. Towards verified artificial intelligence. *Arxiv*, abs/1606.08514, 2016.

[SST18] Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis. Modeling for verification. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 75–105. Springer International Publishing, Cham, 2018.

[SWH11] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles. In Richard N. Taylor, Harald Gall, and Nenad Medvidović, editors, *2011 33rd international conference on software engineering (ICSE 2011)*, page 391, New York, 2011. Curran.

[SWRH10] M. Staats, M.W. Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing: Evaluation of effectiveness. In *NFM '10'*, pages 161–170, Washington D.C., USA, 2010.

[SZ16] Jörg Schaeuffele and Thomas Zurawka. *Automotive Software Engineering - Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen.* Springer Fachmedien GmbH, Wiesbaden, Germany, 6 edition, 4 2016.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* Intelligent robotics and autonomous agents. MIT, Cambridge, Mass. and London, 2005.

[Toe20] Jan Toennemann. Evaluation of a toolchain for model-based development and requirements-based automatic test case generation, 2020.

[TPT$^+$12] Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, Timo Latvala, and Laura Nummila. Formal development and assessment of a reconfigurable on-board satellite system. In Frank Ortmeier and Peter Daniel, editors, *Computer safety, reliability and security*, volume 7612 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 210–222. Springer, Heidelberg, 2012.

[TR05] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[UMR$^+$15] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 982–988, 2015.

[UPÇ12] A. Galip Ulsoy, Huei Peng, and Melih Çakmakci. *Automotive Control Systems.* Cambridge University Press, 2012.

[WBB09] Ralf Wimmer, Bettina Braitling, and Bernd Becker. Counterexample generation for discrete-time markov chains using bounded model checking. In

Neil D. Jones and Markus Müller-Olm, editors, *Verification, model checking, and abstract interpretation*, volume 5403 of *Lecture notes in computer science, 0302-9743*, pages 366–380. Springer, Berlin, 2009.

[WDF+16] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: A case study. *IEEE Transactions on Human-Machine Systems*, 46(2):186–196, 2016.

[WF12] Tichakorn Wongpiromsarn and Emilio Frazzoli. Control of probabilistic systems under dynamic, partially known environments with temporal logic specifications. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 7644–7651. IEEE, 2012.

[WFCJ11] Matt Webster, Michael Fisher, Neil Cameron, and Mike Jump. Formal methods for the certification of autonomous unmanned aircraft systems. In Francesco Flammini, Sandro Bologna, and Valeria Vittorini, editors, *Computer Safety, Reliability, and Security*, volume 6894 of *LNCS sublibrary. SL 2, Programming and software engineering*, pages 228–242. Springer, Heidelberg, 2011.

[WJÁ+14] Ralf Wimmer, Nils Jansen, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. Minimal counterexamples for linear-time probabilistic verification. *Theoretical Computer Science*, 549:61–100, 2014.

[WJV+13] Ralf Wimmer, Nils Jansen, Andreas Vorpahl, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. High-level counterexamples for probabilistic automata. In Kaustubh Joshi, editor, *Quantitative evaluation of systems*, volume 8054 of *LNCS Sublibrary: SL 1 - Theoretical computer science and general issues*, pages 39–54. Springer, Heidelberg, 2013.

[WKLS18] Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, and Shinichi Shiraishi. Runtime monitoring for safety of intelligent vehicles. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 31:1–31:6, New York, NY, USA, 2018. ACM.

[WMB+18] Garrett C. Waycaster, Taiki Matsumura, Volodymyr Bilotkach, Raphael T. Haftka, and Nam H. Kim. Review of regulatory emphasis on transportation safety in the united states, 2002-2009: Public versus private modes. *Risk analysis : an official publication of the Society for Risk Analysis*, 38(5):1085–1101, 2018.

[WOK+16] Mirko Wächter, Simon Ottenhaus, Manfred Kröhnert, Nikolaus Vahrenkamp, and Tamim Asfour. The armarx statechart concept: Graphical programing of robot behavior. *Frontiers in Robotics and AI*, 3, 2016.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 25–36, New York, NY, USA, 2006. Association for Computing Machinery.

[WS05] David P. Watson and David H. Scheidt. Autonomous systems. *Johns Hopkins APL Technical Digest*, (26):368–376, 2005.

[WTM12] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012.

[WUB$^+$12] Tichakorn Wongpiromsarn, Alphan Ulusoy, Calin Belta, Emilio Frazzoli, and Daniela Rus. Incremental temporal logic synthesis of control policies for robots interacting with dynamic agents. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 229–236. IEEE, 10/7/2012 - 10/12/2012.

[WUB$^+$13] Tichakorn Wongpiromsarn, Alphan Ulusoy, Calin Belta, Emilio Frazzoli, and Daniela Rus. Incremental synthesis of control policies for heterogeneous multi-agent systems with linear temporal logic specifications. In *2013 IEEE International Conference on Robotics and Automation*, pages 5011–5018. IEEE, 2013.

[Xu21] Wei Xu. From automation to autonomy and autonomous vehicles. *Interactions*, 28(1):48–53, 2021.

[ZZC05] Shikun Zhou, Hussein Zedan, and Antonio Cau. Run-time analysis of time-critical systems. *J. Syst. Archit.*, 51:331–345, 2005.