Mälardalen University Licentiate Thesis
No.291

Automatic Model Generation and Scalable Verification for Autonomous Vehicles

Mission Planning and Collision Avoidance

Rong Gu

April 2020

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Rong Gu, 2020
ISSN 1651-9256
ISBN 978-91-7485-469-5
Printed by E-Print AB 2020, Stockholm, Sweden
Distribution: Mälardalen University Press

Abstract

Autonomous vehicles such as mobile driverless construction equipment bear the promise of increased safety and industrial productivity by automating repetitive tasks and reducing manual labor costs. These systems are usually involved in safety- or mission-critical scenarios, therefore they require thorough analysis and verification. Traditional approaches such as simulation and prototype testing are limited in their scope of verifying a system that interacts autonomously with an unpredictable environment that assumes the presence of humans and varying site conditions. Methods for formal verification could be more suitable in providing guarantees of safe operation of autonomous vehicles within specified unpredictable environments. However, employing them entails addressing two main challenges: (i) constructing the models of the systems and their environment, and (ii) scaling the verification to the incurred model complexity. We address these two challenges for two essential aspects of autonomous vehicle design: mission planning and collision avoidance. Though inherently different, communication between these two aspects is necessary, as the information obtained from verifying collision avoidance can help to improve the mission planning and vice versa. Finding a solution that addresses both mission planning and collision avoidance modeling and verification, while decoupling them for solution maintainability is one crux of this study. Another one deals with demonstrating the applicability and scalability of the proposed approach on complex and industrial-level systems.

In this thesis, we propose a two-layer framework for mission planning and verification of autonomous vehicles. The framework separates the modeling and computing mission plans in a discrete environment, from the vehicle movement within a continuous environment, in which collision avoidance algorithms based on dipole fields are proven to ensure safe behavior. We call the layer for mission planning, the *static layer*, and the other one the *dynamic layer*. Due to the inherent difference between the layers, we use different mod-

eling and verification approaches, namely: (i) the timed automata formalism and the UPPAAL model checker to compute mission plans for the autonomous vehicles, and (ii) hybrid automata and statistical model checking using UPPAAL Statistical Model Checker to verify collision avoidance and safe operation. We create model-generation algorithms, based on which we develop tool support for the static layer, called TAMAA (Timed-Automata-Based Planner for Autonomous Agents). The tool enables the designers to configure their systems and environments in a graphical user interface, and utilize formal methods and advanced path-planning algorithms to generate mission plans automatically. TAMAA also integrates reinforcement learning with model checking to alleviate the state-space explosion problem when the number of vehicles increases. We create a hybrid model for the dynamic layer of the framework and propose a pattern-based modeling method for the embedded control systems of the autonomous vehicles to ease the design and facilitate reuse. We validate the proposed framework and design method on an industrial use case involving autonomous wheel loaders, for which we verify invariance, reachability, and liveness properties.

Sammanfattning

Autonoma fordon, exempelvis förarlösabyggfordon, lovar ökad säkerhet och industriell produktivitet genom att automatisera upprepade uppgifter och minska manuella arbetskraftskostnader. Dessa system är vanligtvis involverade i säkerhets- eller uppdragskritiska scenarier, därför kräver de noggrann analys och verifiering. Traditionella tillvägagångssätt som simulering och prototyptestning är begränsade till verifiering av system som samverkar autonomt med en oförutsägbar miljö som förutsätter närvaron av människor och olika platsförhållanden. Metoder för formell verifiering kan vara mer lämpade för att garantera säker drift av autonoma fordon i specifierade oförutsägbara miljöer. Att tillämpa dem innebär emellertid två huvudutmaningar: (i) konstruktion av modellerna av systemen och deras miljö, och (ii) skalning av verifieringen till den uppkomna modellkomplexiteten. Vi tar upp dessa två utmaningar inom ramen av två väsentliga aspekter viddesign av autonoma fordon: uppdragsplanering och undvikande av kollision. Trots att de två aspekterna skiljersig åt är kommunikation mellan dessa två aspekter nödvändig, eftersom informationen som erhålls för att verifiera kollisionsundvikande kan bidra till att förbättra uppdragsplaneringen och vice versa. Att hitta en lösning som hanterar både uppdragsplanering och modellering och verifiering av kollisionsundvikande, samtidigt som den frikopplar delarna för att kunna underhålla dem är en svårighet i dessa utmaningar. En annan handlar om att visa huruvida den föreslagna metoden är tillämpbar och skalbar på komplexa och industriella system.

I den här avhandlingen föreslår vi ett ramverk i två lager för uppdragsplanering och verifiering av autonoma fordon. Ramverket skiljer modelleringen och uppdragsplaneringen i en diskret miljö, från fordonets rörelse i en kontinuerlig miljö, där kollisionsundvikelsealgoritmer baserade på dipolfält är bevisade för att säkerställa säkert beteende. Vi kallar lagret för uppdragsplanering, "det statiska lagret" och det andra för "det dynamiska lagret". På grund av den inneboende skillnaden mellan lagren använder vi olika modellerings- och

verifieringsmetoder, nämligen: (i) till det tidsinställda lagret använder vi tidsautomateroch mjukvaran UPPAAL för att beräkna uppdragsplaner för de autonoma fordonen, och (ii) hybridautomater och statistisk modellkontroll med hjälp av UPPAAL Statistical Model Checker för att kontrollera undvikande av kollision och säker drift. Vi skapar modellgenerationsalgoritmer som vi baserar utvecklandet av verktygsstöd för det statiska skiktet på. Verktyget, TAMAA (Timed-Automata-Based Planner for Autonomous Agents), gör det möjligt för designers att konfigurera sina system och miljöer i ett grafiskt användargränssnitt och använda formella metoder och avancerade sökplaneringsalgoritmer för att generera uppdragsplaner automatiskt. TAMAA integrerar också förstärkningslärande för att lindra problemet med exponentiell tillväxt av tillstånd när antalet fordon ökar. Vi skapar en hybridmodell för ramens dynamiska lager och föreslår en mönsterbaserad modelleringsmetod för de inbäddade styrsystemen i autonoma fordonen för att underlätta designen och återanvändning. Vi validerar det föreslagna ramverket och konstruktionsmetoden för ett industriellt användningsfall som involverar autonoma hjullastare, för vilket vi verifierar diverse relevant egenskaper.

致我的父亲母亲

To my parents

吾生也有涯，而知也无涯。
以有涯随无涯，何如？

– 庄子·内篇·养生主

My life has an end.
The universe of knowledge has no end.
How would it be,
to pursue the endless knowledge with a limited life?

– *Chuang Tzu*

Acknowledgments

Three years ago, a boy who had barely left his home country gave up his career in one of the largest avionic institutes in China and came to Sweden to pursue his dream to become a computer scientist. Without knowing anything about the country and language, but owning a zealous heart to learn the interesting knowledge and culture, he started his journey. Now, he is almost half of becoming a PhD of computer science, and sitting in front of his laptop, writing the acknowledgements for his licentiate thesis. Time flies!

During these three years, things were not always easy, sometimes even tough. Imagine how hard it could be for a man who has only used English when taking exams now need to live and work in this language. Luckily, I have a lovely family behind me. My wife was always there for me. We took good care of each other and went through some difficult times altogether. Now life is much brighter and easier, but I will never forget how she encouraged me when I was hesitating and doubting myself. Without her, I could not come to Sweden in the first place and never dream of taking the courage to give up an easy life and pursuing a career I really like. Thank you, Rui. You are my best friend and my love, the one whom I want to share every laugh and tear with.

I want to thank my parents. They always have faith in me. No matter what I decide to do, they support me. I will never forget the scene when they sent me to Sweden at the airport. Chinese people barely hug. They just stood behind the security gate and waved to me. Even after I walked far away, they were still there, watching me go away. Their love is wordless and invaluable. Thank you, mom and dad, for everything you have done for me. I can sense the profound happiness you get in the other end of the phone when I was telling you about my first publication, and I want to dedicate all my achievements to you, for your upbringing and endless love.

I would like to thank my supervisors, Associate Professor Cristina Seceleanu, Professor Kristina Lundqvist, Dr Eduard Enoiu, and former supervi-

sor Dr Raluca Marinescu, for your guidance and support during the journey. Cristina, I want to send my special thanks to you, for your kind heart of tolerating all my faults and waywardness, and patience of teaching me everything repetitively. Your consistent enthusiasm for work and life is and will always be my light for the path. Many thanks to Eddie and Raluca, your advice on how to doing research, writing papers, and giving presentations, are invaluable for me. I really enjoy working with you and playing board games together. Kristina, thank you for being there when I need someone to talk to and sharing your rich experience in academia. Your criticism is always mild and accurate. It is my luck to have you as my co-supervisor.

Being a doctoral student in a project collaborating closely with industries, I was given a lot of opportunities to get a close look at their products and producing lines. Our smooth cooperation with Volvo CE has enormously motivated and inspired my research. I would like to thank Martinsson Torbjörn, for organising the interesting workshops and discussion between us the colleagues in Volvo CE. Your selfless sharing is priceless for us.

I would like to express my deep gratitude to the faculty examiner, Professor Kim Larsen, and the grading committee members: Associate Professor Dilian Gurov, and Adjunct Professor Marina Walden for kindly accepting our invitation and dedicating part of their valuable time to review my study. It is truly my honour to have you as the reviewers of this thesis.

I would like to thank Associate Professor Alessandro Papadopoulos for reviewing the proposal and the initial version of the thesis, and Dr Peter Backeman for the help with the Swedish abstract.

My passion for pursuing a career in academia stems from the zeal for research and education. Many thanks to Mälardalen University for offering me so many opportunities for teaching. I would like to thank Lecturer Afshin Ameri and Professor Mats Björkman for providing the chance to give lectures to our students, and the enormous help and tolerance from you when I was not ready and feeling nervous of standing on the platform for the first time. This experience and your advice would benefit me a lot in my future career.

I still remember vividly that in the first *fika*, a.k.a. coffee break in English, with our colleagues in IDT, Cristina joked that if you know how to *fika*, you would get to know how to become a PhD. Although it is a joke, it turns out to be true to some extend. The interesting stories from all over the world that we shared in the *fika* time everyday enriched my life and gave me many practical suggestions of living in a foreign country. I would like to express my appreciation to my colleagues, Simin, Asha, Francisco, etc., for laughing at my bad jokes and giving me advice in all aspects of life as a doctoral student.

I have also made many close friends in daily life since I moved to Sweden. Danny and Birgitta are the nicest people I have ever met. Their kindness and patience really make me feel warm in heart and never alone. Joakim and Tamara, many thanks for spending so many weekends with us. It is very hard to find someone in a new environment whom you can play with and share happiness and sadness. Luckily we met you. William and Siiri, it is a real pleasure to be neighbours and friends with you. Your attitude towards work and life, and the extreme kindness to everyone influenced me deeply.

Last but not least, to my dear friends whom I have known for many years, Feng Jindong, Gao xinqi, Cheng siyuan, Dong ruixi. The friendship that is established in the youth age is always genuine and invaluable. Though we are apart physically, just like what the beautiful Chinese poem says, a bosom friend afar brings distance near.

Rong Gu
Västerås, April, 2020

List of Publications

Papers Included in the Thesis[1]

Paper A *Formal Verification of an Autonomous Wheel Loader by Model Checking.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Published in Proceedings of the 6$^{\text{th}}$ Conference on Formal Methods in Software Engineering (FormaliSE), ACM, 2018.

Paper B *Towards a Two-Layer Framework for Verifying Autonomous Vehicles.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Published in Proceedings of 11$^{\text{th}}$ Annual NASA Formal Methods Symposium (NFM), Springer, 2019.

Paper C *TAMAA: UPPAAL-based Mission Planning for Autonomous Agents.* Rong Gu, Eduard Enoiu, and Cristina Seceleanu. Published in Proceedings of 35th Symposium On Applied Computing (SAC), ACM, 2020.

Paper D *Combining Model Checking and Reinforcement Learning for Scalable Mission Planning of Autonomous Agents.* Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, MDH-MRTC-330/2020-1-SE, 2020.

[1]The included papers have been reformatted to comply with the thesis layout

Contents

I Thesis **1**

1 Introduction **3**
1.1 Thesis Overview 7

2 Preliminaries **11**
2.1 Timed Automata and UPPAAL 11
2.2 Hybrid Automata and UPPAAL SMC 13
2.3 Path-Planning Algorithms 15
2.4 Dipole Flow Field Algorithm 16
2.5 Reinforcement Learning 17

3 Research Problem **19**
3.1 Problem Description 19
3.2 Research Goals 20

4 Research Methods **23**

5 Thesis Contributions **25**
5.1 A Two-Layer Framework for Modeling and Verification of Autonomous Vehicles 25
5.2 Formal Modeling and Verification of Path-Planning and Collision-Avoidance Algorithms 27
5.3 Scalable Synthesis of Collision-Free Mission Plans Via Model Checking and Reinforcement Learning 29
5.4 Formal Modeling and Verification of the Embedded Control System and Dynamics of Autonomous Vehicles 33
5.5 Validating Our Solution on an Industrial Use Case: Autonomous Wheel Loaders 34

5.6 Research Goals Revisited 38

6 Related Work 41
6.1 Mission Planning for Autonomous Agents 41
6.2 Verification of Autonomous Agents 43

7 Conclusions and Future Work 45
7.1 Limitations 46
7.2 Future Work 47

Bibliography 49

II Included Papers 57

8 Paper A:
Formal Verification of an Autonomous Wheel Loader by Model Checking 59
8.1 Introduction 61
8.2 Autonomous Wheel Loader: Architecture and Requirements . 62
8.3 Preliminaries 66
8.3.1 Timed Automata and UPPAAL 66
8.3.2 A* Algorithm 69
8.3.3 Dipole Flow Field for Collision Avoidance 69
8.4 AWL's Modeling and Verification 70
8.4.1 Map Abstraction 71
8.4.2 Movements Abstraction 72
8.4.3 Formal Model of AWL's Control System 73
8.4.4 AWL's Model Verification 78
8.5 Discussion 83
8.6 Related work 84
8.7 Conclusions 86
Bibliography 87

9 Paper B:
Towards a Two-Layer Framework for Verifying Autonomous Vehicles 91
9.1 Introduction 93
9.2 Preliminaries 94
9.2.1 Hybrid Automata and UPPAAL SMC 94

9.2.2 Theta* Algorithm 95
9.2.3 Dipole Flow Field for Collision Avoidance 96
9.3 Use Case: Autonomous Wheel Loader 97
9.4 A Two-level Framework for Planning and Verifying Autonomous Vehicles 98
9.5 Pattern-based Modeling of the Dynamic Layer 100
9.5.1 Patterns for the Execution Unit 101
9.5.2 Patterns for the Control Unit 102
9.5.3 Encoding the Control Unit Patterns as Hybrid Automata 103
9.6 Use Case Revisited: Applying Our Method on AWL 105
9.6.1 Formal Model of the Control Unit 106
9.6.2 Statistical Model Checking of the AWL Formal Model 107
9.7 Related Work 109
9.8 Conclusions and future work 110
Bibliography 113

10 Paper C:
TAMAA: UPPAAL-based Mission Planning for Autonomous Agents
117
10.1 Introduction 119
10.2 Preliminaries 121
10.2.1 UPPAAL Timed Automata 121
10.3 TAMAA Approach 122
10.3.1 Use Case: Autonomous Wheel Loader 122
10.3.2 Workflow of TAMAA 124
10.3.3 Model Formalization and Definitions of Concepts . . . 125
10.3.4 Automatic Generation of Autonomous Mission Models via TAMAA 131
10.4 TAMAA Implementation and Evaluation 137
10.4.1 Implementation and User Interface 137
10.4.2 Evaluation of TAMAA's Applicability 138
10.4.3 Evaluation of TAMAA's Scalability 139
10.5 Related Work 141
10.6 Conclusions and Future Work 141
Bibliography 143

**11 Paper D:
Combining Model Checking and Reinforcement Learning for Scalable Mission Planning of Autonomous Agents 147**
11.1 Introduction . . . 149
11.2 Preliminaries . . . 151
11.2.1 Timed Automata and UPPAAL . . . 151
11.2.2 UPPAAL STRATEGO . . . 152
11.2.3 Reinforcement Learning . . . 152
11.3 Problem Description . . . 153
11.3.1 Problem Analysis . . . 154
11.3.2 Uncertainties and Scalability of Mission Planning . . . 155
11.4 MCRL: Combining Model Checking and Reinforcement Learning in UPPAAL . . . 156
11.4.1 Timed-Automata-Based Model for Mission Plan Synthesis . . . 157
11.4.2 MCRL Method Description . . . 159
11.5 Experimental Evaluation . . . 165
11.5.1 Discussion . . . 166
11.6 Related Work . . . 170
11.7 Conclusion and Future Work . . . 171
Bibliography . . . 173

I

Thesis

Chapter 1

Introduction

Autonomous vehicles are drawing an increased attention from both researchers and practitioners. The benefits brought by autonomy compel industry and academia to invest a large amount of resources to realize this concept. Industrial machines such as wheel loaders and haulers used in construction sites are equipped with autonomous driving functionality. These systems bear the promise of increased safety and industrial productivity by automating repetitive tasks and reducing labor costs. Such systems are complex, and most often subjected to timing constraints for productivity reasons, hence a thorough verification of their autonomous functionality is crucial, in order to obtain guarantees of their dependable operation.

The environment in which autonomous construction vehicles operate is hazardous, that is, possibly populated with static and dynamic obstacles that need to be discovered and avoided by all means, even in harsh weather conditions. On one hand, such vehicles are designed to perform predefined tasks, and, unlike usual industrial robots, they operate in large construction sites, alongside other machines and humans. On the other hand, the environment is contained and controlled, thus the vehicle's autonomy is bounded.

Traditional approaches such as simulation and prototype testing might not be sufficient for verifying a system that interacts autonomously with an unpredictable environment that assumes the presence of humans and varying site conditions. These techniques are either applied later in the system's development cycle, or they simply cannot prove, exhaustively or statistically, the satisfaction of properties related to autonomous behaviors such as path planning, path following, and collision avoidance. Formal verification [1] could be

therefore applied on design models, to complement the traditional verification techniques, yet being able to verify such complex systems is a big challenge. The complexity of the system stems from the integrated intelligent algorithms, such as those for collision avoidance, as well as the combination of the vehicle's control system and the continuous behavior of the vehicle in motion. Several related studies on motion planning and verification of autonomous vehicles propose a means of decoupling the discrete planning from the hybrid control and demonstrate the applicability of utilizing formal methods in this area [2, 3, 4, 5]. The authors' efforts in motion planning strongly inspire us to address this problem by using formal methods. However, few of them in principle, consider timing requirements and finding a solution for scalable verification. If a model becomes too complex, for instance by assuming a large number of autonomous robots or vehicles, its formal verification by exhaustive model checking might not be feasible due to the well-known state-space explosion problem.

Overall, in this thesis, we address the challenges mentioned above by providing solutions for scalable formal analysis (exhaustively when possible and statistically in other cases) of autonomous vehicle behavior with respect to mission planning, path following, and collision avoidance. We also look into the design of the embedded control system of the vehicles, which is a distributed system consisting of several units. Additionally, our solutions provide a means to automatically generate formal models amendable to formal analysis, from high-level descriptions specified by designers in a GUI called MMT, and a pattern-based modeling method for the hybrid model describing the continuous movement of the vehicles, which aims at providing an ability of verification in a realistic environment model.

We start our research by studying a use case provided by Volvo CE, a leading manufacturer of construction equipment. The use case focuses on autonomous wheel loaders (AWL) that are used in construction sites to perform operations without human intervention. As an example, in Figure 1.1 we show the case of an AWL that is utilized to transport materials in a quarry site. According to the requirements from Volvo CE, an AWL digs a given stone pile and carries an amount of stones to a primary crusher that crushes the stones at given fractions, after which the vehicle unloads the stones onto the conveyor belt. Next, the AWL moves to the other end of the primary crusher and loads the crushed stones. It then continues moving to the secondary crusher to unload the stones and finishes its one-round job. During this process, the AWL carries out its tasks autonomously and moves to the charging point when its battery level is low. The AWL has to also avoid static obstacles (e.g., holes and

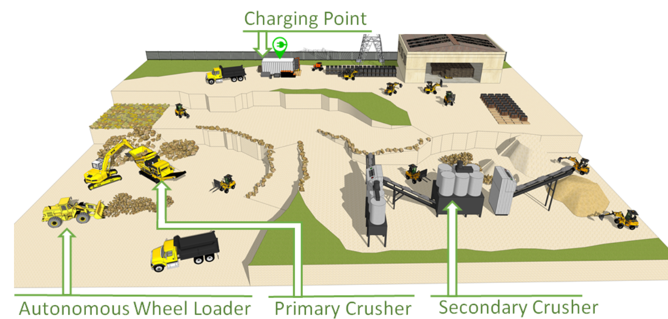


Figure 1.1: An example of a quarry for an autonomous wheel loader

rocks above a certain size, existing on the ground) as well as possible dynamic obstacles (e.g., other mobile machines or humans). Hence, the design of AWL involves mission planning, path following, and collision avoidance.

Designing such a system poses two main challenges, as it contains two aspects that are inherently different. The first challenge is about path planning and task scheduling, and the other one is about the verification of the AWL in a continuous environment. The former aspect does not concern the continuous features of the AWL as it only focuses on making plans that guide the AWL towards the destination, and on carrying out certain tasks, in a certain order, at given positions called milestones, within prescribed amounts of time. Therefore, we can describe the environment as a discrete Cartesian grid, which facilitates modeling the path-planning algorithms [6]. However, verifying collision avoidance requires a continuous environment, in which the kinematics of an AWL can be captured. Since the mission planning phase considers only static obstacles, the possibly unpredictable movement of existing dynamic obstacles might cause the AWL to deviate too much from its originally planned path in order to avoid them, triggering a re-plan. Once computed, the new plan has to be verified again in the assumed continuous environment. The iteration continues until a verified safe and efficient mission plan is generated. Therefore, in order to design a safe AWL, we need to propose a modeling and verification solution that decouples the discrete part from the continuous part, in order to facilitate reuse and ease of change, yet allow bi-directional communication. This increases the complexity of the problem and leads to the second challenge: applicability and scalability of our approach. Assuming the approach is adopted in industrial systems, where the environment is large, or the number of AWL or the missions of AWL increases, our method should be

able to still find solutions in reasonable time.

To meet the above needs, in this thesis we propose a two-layer framework consisting of a *static* and a *dynamic* layer, respectively, between which data is exchanged according to a chosen communication protocol [7, 8]. The *static layer* is responsible for path and mission planning for the autonomous vehicles, according to possibly incomplete information of the environment. In this layer, known static obstacles are assumed, together with milestones representing points of operation of the autonomous vehicles. A* [9] and Theta* [6] algorithms for path planing are modeled and verified in this layer. The *dynamic layer* is dedicated to simulating and verifying the system that follows autonomously the reference path from the starting point to destination, generated by the static layer, while considering continuous motion in an environment that contains moving and unforeseen obstacles. Hence, a collision-avoidance algorithm based on the dipole field [10] is encoded in the model used in this layer. The structure of the framework relies on the well-known design principle of separation of concerns: it separates the static high-level path planning that assumes an environment with a predefined sequence of milestones that need to be reached, as well as static obstacles, from the dynamic functions like collision avoidance, thus providing a separation of concerns for the system design, modeling, and verification. The specific contributions in each layer of the framework are described as below:

i) **Static Layer**. We build the model of the static layer by using timed automata and verify it exhaustively by employing the state-of-the-art model checker called UPPAAL [11]. The main concepts at this level, such as vehicle movement, tasks execution, and monitors for events are formally defined in our work [12], where we also present the tool that supports the static layer modeling and analysis. These definitions are the foundation of the model generation algorithms, which are programmed in the tool of the static layer called **TAMAA (Timed-Automata-based planner for Multiple Autonomous Agents)**. Furthermore, to solve the model checking scalability problem incurred by the increased number of agents, we propose an innovative method that combines **reinforcement learning** [13] with the **model-checking technique**, namely **MCRL**.

ii) **Dynamic Layer**. As timed automata do not support modeling the continuous movement, we design the model of the linear movement and rotation of the autonomous vehicles by using hybrid automata. Due to the undecidability of verifying most properties of hybrid automata and the uncertainty of environment events, we use UPPAAL Statistical Model Checker (UPPAAL SMC) for verification. To facilitate the modeling of the complex embedded control

software of the autonomous vehicles and reuse of the model, we propose a **pattern-based method** to describe the processes and functions in the embedded control software, formally, as timed automata with uniform distribution of the discrete actions, and uniform or exponential distributions for the delay actions. We adopt statistical model checking to verify the model of the dynamic layer and discover several critical scenarios that bear the potential to cause collisions, due to the limitation of the collision-avoidance algorithm, which are reported in our paper [8]. The methods are evaluated in an industrial use case: the autonomous wheel loader, provided by Volvo CE. In summary, with the help of our solution, designers are able to synthesize mission plans for autonomous vehicles by simply configuring the environment and tasks for them in a GUI. The synthesized mission plans are formally verified against various requirements, including timing constraints. The method also alleviates the state-space-explosion problem when the number of vehicles raises so that it is applicable and scalable for industrial use cases.

1.1 Thesis Overview

This thesis is divided into two parts. The first part is a summary of our research, including the preliminaries of this thesis (Chapter 2), the problem formulation and our research goals (Chapter 3), the research methods applied in this thesis (Chapter 4), a brief overview of our contributions (Chapter 5), a discussion on the related work (Chapter 6), as well as our conclusions, limitations and future work directions (Chapter 7).

The second part is a collection of papers included in this thesis, listed as follows:

Paper A *Formal Verification of an Autonomous Wheel Loader by Model Checking.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, Kristina Lundqvist. In Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE), ACM, 2018.

Abstract: In an attempt to increase productivity and the workers' safety, the construction industry is moving towards autonomous construction sites, where various construction machines operate without human intervention. In order to perform their tasks autonomously, the machines are equipped with different features, such as position localization, human and obstacle detection, collision avoidance, etc. Such systems are safety critical, and should operate

autonomously with very high dependability (e.g., by meeting task deadlines, avoiding (fatal) accidents at all costs, etc.). An Autonomous Wheel Loader is a machine that transports materials within the construction site without a human in the cab. To check the dependability of the loader, in this paper we provide a timed automata description of the vehicle's control system, including the abstracted path planning and collision avoidance algorithms used to navigate the loader, and we model check the encoding in UPPAAL, against various functional, timing and safety requirements. The complex nature of the navigation algorithms makes the loader's abstract modeling and the verification very challenging. Our work shows that exhaustive verification techniques can be applied early in the development of autonomous systems, to enable finding potential design errors that would incur increased costs if discovered later.

My contribution: I was the primary driver of the paper, developed the method, wrote most of the text, and performed all the modeling and verification activities. The other authors contributed with valuable ideas and comments.

Paper B *Towards a Two-Layer Framework for Verifying Autonomous Vehicles.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. In *Proceedings of the 11th Annual NASA Formal Methods Symposium (NFM), Springer, 2019.*

Abstract: Autonomous vehicles rely heavily on intelligent algorithms for path planning and collision avoidance, and their functionality and dependability can be ensured through formal verification. To facilitate the verification, it is beneficial to decouple the static high-level planning from the dynamic functions like collision avoidance. In this paper, we propose a conceptual two-layer framework for verifying autonomous vehicles, which consists of a static layer and a dynamic layer. We focus concretely on modeling and verifying the dynamic layer using hybrid automata and UPPAAL SMC, where a continuous movement of the vehicle as well as collision avoidance via a dipole flow field algorithm are considered. In our framework, decoupling is achieved by separating the verification of the vehicle's autonomous path planning from that of the vehicle autonomous operation in its continuous dynamic environment. To simplify the modeling process, we propose a pattern-based design method, where patterns are expressed as hybrid automata. We demonstrate the applicability of the dynamic layer of our framework on an industrial prototype of an autonomous wheel loader.

My contribution: I was the main driver of the paper, wrote most of the text and implemented the model, and performed the case study. The other authors contributed with valuable ideas and comments.

Paper C *TAMAA: UPPAAL-based Mission Planning for Autonomous Agents.* Rong Gu, Eduard Enoiu, and Cristina Seceleanu. In *Proceedings of the 35th ACM/SIGAPP Symposium On Applied Computing (SAC)*, ACM, 2020.

Abstract: Autonomous vehicles, such as construction machines, operate in hazardous environments, while being required to function at high productivity. To meet both safety and productivity, planning obstacle-avoiding routes in an efficient and effective manner is of primary importance, especially when relying on autonomous vehicles to safely perform their missions. This work explores the use of model checking for the automatic generation of mission plans for autonomous vehicles, which are guaranteed to meet certain functional and extra-functional requirements (e.g., timing). We propose modeling of autonomous vehicles as agents in timed automata together with monitors for supervising their behavior in time (e.g., battery level). We automate this approach by implementing it in a tool called TAMAA (Timed-Automata-based Planner for Multiple Autonomous Agents) and integrating it with a mission-configuration tool. We demonstrate the applicability of our approach on an industrial autonomous wheel loader use case.

My contribution: I was the main driver of the paper, wrote most of the text, built the model, and conducted the evaluation. The other two authors contributed with valuable ideas and comments.

Paper D *Combining Model Checking and Reinforcement Learning for Scalable Mission Planning of Autonomous Agents.* Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, MDH-MRTC-330/2020-1-SE, 2020. Submitted to FMICS 2020.

Abstract: The problem of mission planning for multiple autonomous agents, including path planning and task scheduling, is often complex, especially when the number of agents grows or requirements include real-time constraints. In this paper, we propose a novel approach called MCRL that integrates model checking and reinforcement learning to overcome this difficulty. Our approach

employs timed automata and timed computation tree logic to describe the autonomous agents' behavior and requirements, and trains the model by a reinforcement learning algorithm, namely Q-learning, to populate a table used to restrict the state space of the model. Our method provides a means to synthesize mission plans for autonomous systems whose complexity exceeds the scalability boundaries of exhaustive model checking, but also to analyze and verify synthesized mission plans to ensure given requirements. We evaluate the proposed method on various scenarios involving autonomous agents, as well as present comparisons with other methods and tools.

My contribution: I was the main driver of the paper, wrote most of the text, implemented algorithms, built the model, and conducted the evaluation. The other authors contributed with valuable ideas and comments.

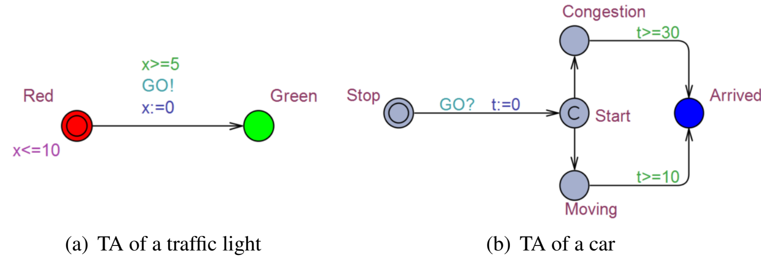
Chapter 2

Preliminaries

In this section, we overview the background information needed for the rest of the thesis: timed automata, hybrid automata, and UPPAAL (SMC), as well as the Theta* and dipole flow field algorithms used for the automatic path generation, and collision avoidance, respectively. We also describe briefly the Q-learning algorithm employed in our reinforcement-learning-aided formal verification.

2.1 Timed Automata and UPPAAL

In this thesis, we use the *timed automata* (TA) [14] to model the movement and task execution of the autonomous vehicles in the static layer, and the UPPAAL model checker [11] to verify the synthesized mission plans. Model checking is the technique that we use in this thesis to traverse the state space of the model and check if it satisfies some properties written in temporal logic. Our choice is justified by the fact that timed automata is an expressive formalism intended to describe the behavior of timed systems in a continuous-time domain. Moreover, the framework is supported by the UPPAAL tool, the state-of-the-art model checker for real-time systems. Timed automata (TA) [14] are finite-state automata extended with real-valued clock variables that measure the elapse of time. UPPAAL uses an extension of TA, called UPPAAL TA henceforth, as the input modeling language; UPPAAL TA extends TA with discrete variables, as well as other modeling features, like urgent and committed locations, synchronization channels, etc. A TA consists of a finite set of locations (represented

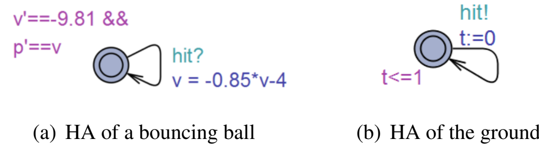


(a) TA of a traffic light (b) TA of a car

Figure 2.1: An example of TA in UPPAAL

graphically as circles), which are connected by directed lines called edges. A pair with a location and a clock valuation is called a state of a TA. Multiple UPPAAL TA can form a Network of Timed Automata (NTA) via parallel composition ("||") [15], by which individual TA are allowed to carry out internal actions (i.e., interleaving), while pairs of TA can communicate via channels, or shared variables. The locations of all automata, together with the clock valuations, define the state of a NTA.

We illustrate the basics of UPPAAL TA via an example. For more details, we refer to the literature [11]. In Figure 2.1, we show an NTA that models a simple "car-traffic light" scenario. Figure 2.1(a) models the behavior of the traffic light, where x is a clock variable that measures the elapse of time and progresses continuously. The traffic light TA has two *locations*, namely *Red* and *Green* (out of which *Red* is the initial location), and an *edge* connecting them. The car TA contains a special *location* named *Start*, which is a committed *location*. *Locations* in UPPAAL TA can be urgent or committed. When an automaton reaches an urgent *location*, marked as "U", it must take the next transition without any delay in time. A committed *location*, marked as "C", indicates that no delay occurs on this *location* and the following transitions from this *location* will be taken immediately. When an automaton is at a committed *location*, another automaton may not take any transitions, unless it is also at a committed *location*. For a generic TA, at each *location* it may non-deterministically choose to: (i) stay and let time elapse as long as the invariant, which is a conjunction of clock constraints associated to the location, is satisfied; (ii) take a transition via an *edge* to another *location*, as long as the guard on the *edge*, which is a conjunction of constrains on discrete variables or clock variables, is satisfied. In Figure 2.1(a), the traffic light TA can stay at *location Red* until x reaches 10, or transfer to *location Green* when x exceeds 5. When moving from *Red* to *Green*, the traffic light TA synchronizes with the car TA in Figure 2.1(b), via a *channel* called *GO*. An exclamation mark "!" follow-



(a) HA of a bouncing ball (b) HA of the ground

Figure 2.2: An example of a hybrid automaton

ing the channel name denotes the sender, and a question mark "?" denotes the receiver. Meanwhile, an assignment on the *edge* is performed to reset the clock, e.g., from *Red* to *Green*, the traffic light TA sets the clock variable *x* to 0. The assignment can also be a function written in a subset of the C language, updating clock variables as well as discrete variables.

The UPPAAL model checker supports the verification of queries written in a decidable subset of Timed Computation Tree Logic (TCTL) [11]. The syntax of a TCTL formula consists of quantifiers over paths and path-specific temporal operators. There are two types of path quantifiers: the universal one, "A" meaning "for all paths", and the existential one, "E" denoting "there exists a path". We are interested in two path-specific temporal operators, that is, "$Always$" ($\Box$) temporal operator meaning that a given formula is true in all states of a path, and the "$Eventually$" ($\Diamond$) operator meaning that a formula becomes true in finite time, in some state along a path. The UPPAAL queries that we verify in this thesis are properties of the form: (i) **Invariance**: $A\Box p$ means that for all paths, for all states in each path, p is satisfied, (ii) **Liveness**: $A\Diamond p$ means that for all paths, p is satisfied by at least one state in each path, (iii) **Reachability**: $E\Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (iv) **Time-bounded Leads to**: $p \leadsto_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter; it is equivalent to the property: $A\,\Box\,(p \Rightarrow A\,\Diamond_{\leq t}\,q)$.

2.2 Hybrid Automata and UPPAAL SMC

UPPAAL SMC [16] is an extension of the tool UPPAAL, which supports statistical model checking of hybrid automata (HA). Instead of exhaustively exploring the state space of the model, statistical model checking randomly executes the model with respect to a given property and apply statistical analysis to estimate the satisfaction of that property. HA in UPPAAL SMC are similar to UPPAAL TA, and extend the latter with a set of continuous variables whose derivatives are described by ordinary differential equations (ODE). Similarly, we illustrate

the basics of HA via a simple example shown in Figure 2.2. For more details, we refer to the literature [17]. Figure 2.2(a) describes the behavior of a bouncing ball via a HA with only one *location*. The HA uses two continuous (real-valued) variables, v and p, denoting the velocity and the position of the ball, respectively. Based on Newtonian laws of motion, the derivative of v is -9.81, which is the minus of the acceleration of gravity, and the derivative of p is the value of velocity. Hence, the invariant of the automaton's *location* is a conjunction of the ODE $v' == -9.81$ and $p' == v$. In Figure 2.2(b), the HA of the ground is modeled, which synchronizes with the bouncing ball HA via *channel hit*. The definition of *channels* in HA is the same as the ones in TA. Hence, this example means that ground HA sends a signal *hit* to the ball HA when the value of the clock variable *t* is less or equal to one, when the assignment of the velocity in Figure 2.2(a) is also executed to change the value and direction of the speed.

In UPPAAL SMC, the HA have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions, and (ii) the non-deterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or (user-defined) exponential distributions for unbounded delays. For example, in Figure 2.2(b), the transition along the self-loop edge is taken following a uniform distribution within the time bound zero to one. In this thesis, only the default uniform distributions for time-bounded delays are used. A model in UPPAAL SMC is a network of HA that communicate via broadcast channels and global variables. Only broadcast channels are allowed for a clean semantics of purely non-blocking automata (automata that are not blocked by synchronized transitions), since the participating HA repeatedly race against each other, that is, they independently and stochastically decide on their own how much to delay before delivering the output, with the "winner" being the automaton that chooses the minimum delay.

UPPAAL SMC supports an extension of *weighted metric temporal logic* for probability estimation, whose queries are formulated as follows: `Pr[bound] (ap)`, where `bound` is the simulation time, `ap` is the formula that supports two temporal operators: "$Eventually$" ($\Diamond$) and "$Always$" ($\Box$). Such queries estimate the probability that `ap` is satisfied within the simulation time bound. Probability comparison (`Pr[bound]`(ψ_1) $\geq$ `Pr[bound]`(ψ_2)) and hypothesis testing (`Pr[bound]`(ψ) $\geq p_0$) are also supported.

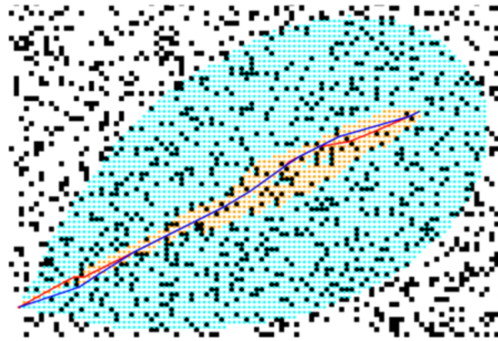


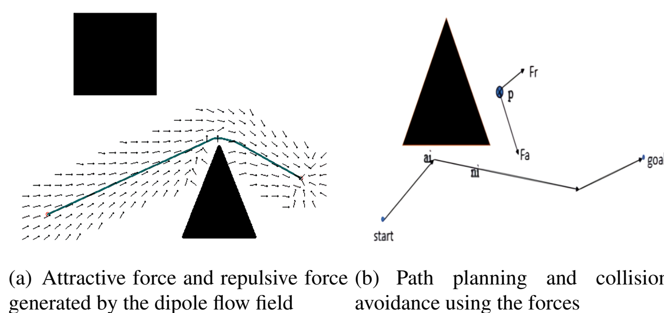
Figure 2.3: A path calculated by Theta* algorithm [6]

2.3 Path-Planning Algorithms

In this thesis, we employ the A* and Theta* algorithms to generate an initial path for our autonomous wheel loader.

The A* algorithm is a widely used algorithm for path finding and graph traversal [9], and it was first introduced by Hart et al. [18]. It is an extension of Dijkstra's algorithm that uses a heuristic function to guide the graph traversal in order to achieve better performance. The basic idea of the A* algorithm is to find a lowest cost path from all possible paths to the destination, similar to Dijkstra's algorithm. While exploring the graph, the cost of the current node is calculated by the following function: $f(n) = g(n) + h(n)$, where n is the current node, $g(n)$ is the cost from the starting node to n, and $h(n)$ is the estimated lowest cost from n to the destination. Intuitively, the A* algorithm aims to find the path that minimizes $f(n)$.

The Theta* algorithm has been firstly proposed by Nash et al. [6] to generate smooth paths with few turns, from the starting position to the destination, for a group of autonomous agents. A path calculated by Theta* that avoids static obstacles and reaches the destination is shown in Figure 2.3. Similar to A* algorithm, the Theta* algorithm explores the map and calculates the cost of nodes also by the function $f(n) = g(n) + h(n)$. In this thesis, we use Manhattan distance [19] for $h(n)$. In each search iteration, the node with the lowest cost among the nodes that have been explored is selected, and its reachable neighbors are also explored by calculating their costs. The iteration is eventually ended if the destination is found or all reachable nodes have been explored. As an optimized version of A* algorithm, Theta* determines the preceding node of a node to be any node in the searching space instead of only



(a) Attractive force and repulsive force generated by the dipole flow field

(b) Path planning and collision avoidance using the forces

Figure 2.4: Demonstration of the dipole flow field algorithm [10]

neighbor nodes. In addition, Theta* adds a line-of-sight (LOS) detection to each search iteration to find an any-angle path that is less zigzagged than those generated by A* and its variants (see Figure 2.3). For the detailed description of the algorithm, we refer the reader to the literature [6].

2.4 Dipole Flow Field Algorithm

Searching for a path from the starting point to the goal point, assuming a large environment, is not an easy task and it is usually computationally intensive. Hence, some studies have adopted methods to generate a small deviation from the initial path, which is much easier to compute than an entirely new path, while being able to avoid obstacles. To avoid collisions, Trinh et al. propose an approach to calculate the *static flow field* for all objects, and the *dynamic dipole field* for the moving objects in the environment [10], which we encode in our formal model of the AWL. In the theory of dynamic dipole field, every object is assumed to be a source of magnetic dipole field, in which the magnetic moment is aligned with the moving direction, and the magnitude of the magnetic moment is proportional to the velocity. In this approach, the static flow field is created within the neighborhood of the initial path generated by the Theta* algorithm. The flow field force is a combination of the attractive force drawing the autonomous wheel loader to the initial path, and the repulsive force pushing it away from obstacles. Unlike the dipole field force, the flow field force always exists, regardless of whether the vehicle is moving or not. As soon as the vehicle equipped with this algorithm gets close enough to a moving obstacle, the magnetic moment around the objects keeps them away

from each other.

Figure 2.4(a) illustrates the attractive force generated by the static flow field, which draws the vehicle to the destination, and the repulsive force generated by the static flow field as well as the dynamic dipole field, which pushes the vehicle away from static and moving obstacles. The combination of the static flow field and the dynamic dipole field ensures that the vehicle moves safely by avoiding all kinds of obstacles and that it eventually reaches the destination, as long as a safe path exists. As it is depicted in Figure 2.4(b), once the vehicle deviates from the initial path caused by the repulsive force of an obstacle, the attractive force can always draw it back to the initial planned path. Compared with other methods [20][21], this algorithm provides a novel method for path planning of mobile agents, in the shared working environment of humans and agents, which suits our requirements well. However, one needs to verify that the algorithm is able to safely avoid all moving obstacles, including unforeseen ones. We carry out this verification as part of our thesis contribution [7, 8] that is described later.

2.5 Reinforcement Learning

Reinforcement learning is a branch of machine learning aiming to calculate how agents should take actions in an environment, in order to maximize the accumulated reward obtained from the environment [13]. In this thesis, we use one of the model-free reinforcement learning algorithms called *Q-learning* [22], which is usually adopted to learn policies that indicate agents the actions to take at different states. A policy is associated with a state action value function called *Q function*, where “Q” stands for “quality”. The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \max_{a'} q^*(s',a')], \quad (2.1)$$

where $q^*(s,a)$ represents the expected reward of executing action a at state s, $\mathbb{E}$ denotes the expected value function, $R(s,a)$ is the reward obtained by taking the action a at state s, γ is a discounting value, s' is the new state resulting from state s by taking action a, $\max_{a'} q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair (s',a'). The equation means that the expected reward of the state-action pair (s,a) is the sum of the current reward and the discounted maximum future reward. As the learning process iterates, the Q-value of each state-action pair converges to the maximum Q-value, i.e., q^*, and the parameters are updated using gradient descent

[23]. Although Q-learning is a model-free algorithm, the learning process often relies on a simulation environment that depends on the form of the model. In this thesis, we utilize the simulation function in UPPAAL to gather the information of state-action pairs of the model, and invoke the Q-learning algorithm in Java code to populate the Q-table storing the rewards of state-action pairs [24], as it is presented in Chapter 11.

Chapter 3

Research Problem

In this chapter, we formulate our research problem and research goals addressed in this thesis.

3.1 Problem Description

The broad focus of this thesis is on formal verification of autonomous vehicles, as well as the automatic generation of formal models for verification. Concretely, we target two aspects, that is, computing missions for the vehicles, and ensuring collision avoidance of all obstacles, static or dynamic ones. The former aspect only aims at making plans that guide the autonomous vehicles towards the destination and execute various tasks complying with some certain rules, such as execution order and timing requirements, whereas the latter aspect focuses on the kinematics of the autonomous vehicles and moving obstacles. Therefore, when making mission plans, one can consider the discrete feature of the environment and the systems. More specifically, only some important positions, a.k.a. milestones, where tasks are carried out, are extracted from the environment and modeled. Tasks are also interpreted as discrete entities rather than continuous operations. Though the number of states of the discrete model is countable, the computation time of synthesizing a mission plan increases exponentially as the number of vehicles increases, because the mission scheduling problem is NP-hard [25].

In summary, the **mission planning problem** is formulated as: how to calculate path plans to guide multiple autonomous vehicles to reach all the mile-

stones in the environment, and schedule the execution of tasks so that the vehicles obey the execution order of tasks and timing requirements when carrying out their missions? Given the complexity of computation, finding a **scalable method** is a related **research problem** in our focus.

The other aspect of the problem deals with **collision avoidance**, where we take into account the continuous behavior and unforeseen moving obstacles. The most important factor considered in this sub-problem is the modeling and verification of the hybrid model of the vehicles as well as their embedded control systems. Therefore, how comprehensively and faithfully the model reflects the real scenarios is one important factor in this sub-problem, whereas the ability of verification must still be preserved by the solution as we focus on generating a correctness- and safety-guaranteed design of autonomous vehicles. In summary, this aspect of the problem is related to finding means to verify the autonomous vehicles' functionalities of following the reference paths and avoiding moving obstacles while they are executing the missions.

The overall problem is the combination of the two aspects and is presented as follows: how to model and verify the mission-planning and collision-avoidance functionalities of multiple autonomous vehicles so that the design of the vehicles is correct and safety-guaranteed, and scalability tamed as the number of vehicles increases?

3.2 Research Goals

To solve the research problem of the thesis as formulated in Section 3.1, we formulate the main research goal of the thesis as follows:

Overall goal. Facilitate the assured design of the embedded control software of autonomous vehicles, with respect to mission planning, as well as path following and collision avoidance functions, by employing formal methods.

A key principle of facilitating the design of software in general is the notion of separation of concerns [26]. The key idea is that one should avoid co-locating different concerns within the design. In our research problem, mission planning does not concern the autonomous vehicles' operations in a continuous dynamic environment, and thus the environment can be abstracted as a discrete model at this level. Therefore achieving the separation of concerns of mission planning and continuous movement including collision avoidance of dynamic obstacles, while backing them by formal verification techniques, could be ben-

eficial. Consequently, the first subgoal that contributes to achieving the broader overall goal is as follows:

Subgoal 1. Provide a means that decouples the design of mission planning from the vehicle's autonomous operation in a continuous dynamic environment, supported by model checking techniques.

The path-planning algorithms employed in this thesis are A* [9] and Theta* [6], and the collision-avoidance algorithm relies on the dipole field concept [10]. When applied in the context of the autonomous system's design, their correctness (meeting provided requirements) needs to be demonstrated. To achieve this, we formulate the second subgoal as below:

Subgoal 2. Ensure the correctness of path-planning and collision-avoidance algorithms, within the context of autonomous wheel loaders.

Mission planning for multiple autonomous vehicles is an NP-hard problem [25] and our problem also involves path planning and uncertainties of tasks execution times, etc. The third subgoal aims to provide a scalable method for mission planning, including path planning and mission scheduling. The subgoal is formulated as follows:

Subgoal 3. Provide scalable synthesis of collision-free static mission plans guaranteed to satisfy given temporal requirements among tasks.

As the synthesized mission plans only consider a part of the environment, such as the identified static obstacles, when the vehicles start to execute the mission plans, they might not only encounter some unforeseen static obstacles but also moving obstacles. Moreover, the dynamics of the motion of the vehicles can also cause uncertainties that have not been considered in the mission-planning phase. Based on these issues, the fourth subgoal is presented as follows:

Subgoal 4. Ensure that the model execution of the vehicle's movement in a dynamic environment fulfills the specified functional, precedence, and timing requirements.

Our solutions need to be validated on industrial use cases, in order to show their applicability in real scenarios. Hence, our fifth subgoal is:

Subgoal 5. Assess the applicability of the proposed synthesis and verification methods on an industrial use case.

In summary, the five subgoals concern different aspects of the research problem, and addressing them enables meeting the overall goal of the thesis. If addressed, subgoal 1 facilitates the design of the embedded control software for autonomous vehicles, by decoupling the two aspects of the problem clearly and initializing the idea of a two-layer framework. Subgoals 2, 3, and 4 focus on different layers and algorithms of the framework and transform the initial high-level idea into a concrete solution. Subgoal 5 serves as an evaluation of the method on an industrial prototype. Altogether, the five subgoals provide a detailed decomposition of the overall goal and clarify the scope and focus of this study.

Chapter 4

Research Methods

In this chapter we introduce the methods that we use to conduct our research in order to address the research goals. We first describe the general process that we follow in our research, after which we explain the concrete methods used in this thesis.

Our research process is shown in Figure 4.1. This research is initiated by industrial problems that have not been solved by industrial solutions nor thoroughly studied by academic researchers. Through a number of visits and discussions with our industrial partner Volvo CE, and thorough analysis of their design artifacts, the outstanding industrial problems are accurately identified. Based on them and also on our analysis of the relevant state-of-the-art, we formulate the research goals, as presented in Section 3.2. The process of identifying and defining research goals is iterative, that is, they are gradually refined throughout the entire study.

To address the formulated research goals, we start by investigating the formal methods and their applications on autonomous vehicles that have been studied by industry and academia. Although we have not yet published the results of our analysis as a standalone paper, we have gotten a general picture of the research area that focuses on applying formal methods in the development of autonomous vehicles.

As a next step, we propose a systematic approach, and design algorithms to facilitate the approach, which address our research goals and are applicable to industrial systems. To implement and evaluate the proposed approach, a set of established research methods are used and reported in the papers included in this thesis as listed in Section 1.1. In Paper A [7], which is also presented in

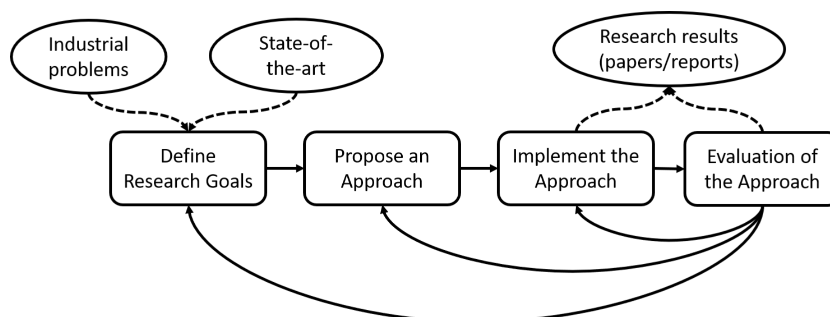


Figure 4.1: Research Process

Chapter 8, and Paper B [8], which is also presented in Chapter 9, we use *case study* research method [27, 28] to demonstrate the correctness and applicability of the proposed methods and identify shortcomings of the path-planning and collision-avoidance algorithms. These studies follow the principle of the *proof of concepts* method [29] to ensure the feasibility of the methods. In Paper C [12], which is also presented in Chapter 10, and Paper D [24], which is included in this thesis as Chapter 11, methods TAMAA and MCRL are implemented as a tool. We conduct experiments to evaluate the methods in different scenarios and make a comparison with other methods. In these studies, we apply the *proof by demonstration* method [29] by developing a tool that supports our TAMAA approach, and evaluate it in an industrial setting. Thanks to the close cooperation with our industrial partner, throughout the entire research process, we perform validation of the research results in experimental settings extracted from real-world scenarios. Such validation brings us the following benefits: i) real-time feedback of whether the methods can solve the identified industrial problems, and ii) assessment of the scalability of the methods conducted on the actual industrial systems.

In the evaluation phase, whenever the results match what we expect in our research goal, we conclude that the goal is achieved, or else we use the obtained experience to propose a new and improved solution to the same problem. Whenever the results differ from our expectations, either for better or worse, we analyze the reasons behind the deviation. After that, if the results are good, we move on to new research goals, whereas if the results are worse than expected, we go back to study literature and propose a new solution, sometimes we also survey, refine, and narrow the research problems and goals.

Chapter 5

Thesis Contributions

In this chapter, we present the contributions of this thesis, which address the aforementioned research goals. We first introduce a two-layer framework for modeling and verifying autonomous vehicles, which acts as our assured design methodology that addresses the overall research goal. Next, we present a development process that support the proposed methodology, and address the subgoals, respectively.

5.1 A Two-Layer Framework for Modeling and Verification of Autonomous Vehicles

In this thesis, we consider two main functionalities of the autonomous vehicles, which have to be realized: mission planning and executing missions autonomously and safely. They involve two levels of concerns, planning and executing. The former includes path planning and task scheduling, which we call *mission planning* in all. When computing mission plans, the system focuses on generating the paths and the order of executing tasks. Autonomous vehicles are designed to synthesize mission plans that satisfy various requirements, e.g., autonomous wheel loaders should carry all the stones to the primary crusher that outputs crushed stones at given fractions, before carrying the crushed stones to the secondary crusher, all within 15 hours. At this level of design, the system model should not include the concrete movement and operation of the vehicles. The vehicles only need to know what to do or where to go at certain time points and positions in the environment.

At the lower level of design, when the vehicles start to execute the synthesized plans, their behaviors have to be modeled in a continuous manner so that the models are as realistic as possible. Hence, behaviors like accelerating, uniform movement, decelerating, and specific motions, e.g., digging, loading, etc. have to be considered and modeled at this level. Therefore, mission planing and the autonomous operation in a continuous environment can be decoupled to reduce the complexity of the design and provide a separation of concerns.

In **Paper B** [8], which is also included as **Chapter 9** in this thesis, we propose an initial design of a two-layer framework consisting of a *static layer* and a *dynamic layer*, which is depicted in Figure 5.1. The communication protocol supports data exchange between the layers. The *static layer* is responsible for path and mission planning, based on the information of the environment detected by the *dynamic layer*. The static layer includes static obstacles and milestones where the tasks should be carried out. Moving obstacles that are unforeseen by the autonomous vehicles are considered in the *dynamic layer*, which is designed to simulate and verify the systems to guarantee that they follow the reference path generated by the *static layer* and avoid dynamic obstacles. These two layers support the modeling of mission planning and modeling of the continuous behavior separately, such that the desired decoupling is achieved.

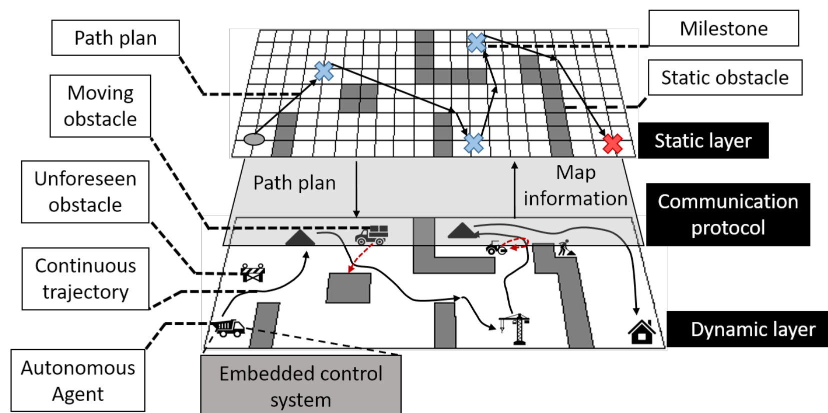


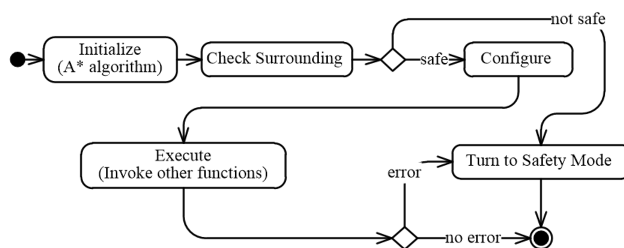
Figure 5.1: A two-layer framework for planning and verifying autonomous vehicles

5.2 Formal Modeling and Verification of Path-Planning and Collision-Avoidance Algorithms

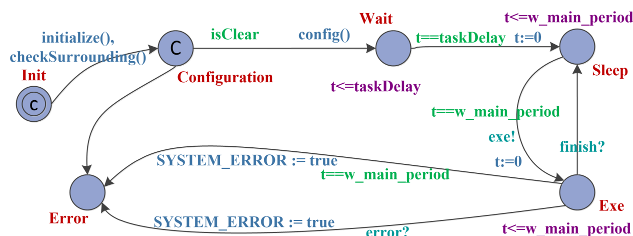
The path-planning and collision-avoidance algorithms are adopted in safety- and mission-critical systems, thus it is crucial to ensure the correctness of these algorithms in the context of autonomous vehicles. Two types of path-planning algorithms are considered in this thesis, one is A* algorithm that is an extension of Dijkstra's algorithm and uses a heuristic function to guide the graph traversal [9]. The other one is the Theta* algorithm that calculates smooth paths with fewer turning points of any angles. When the paths are calculated, the autonomous vehicles need to be able to follow the path, while avoiding all kinds of obstacles on the path. The algorithm for collision avoidance that is adopted in our model is proposed by Trinh et al. [10], and relies on the concept of dipole flow field, which consists of static flow field and dynamic dipole field. The former is calculated for all objects in the environment so that the vehicles can navigate themselves following the reference path and avoiding static obstacles. The latter assumes all moving objects to be sources of magnetic dipole fields and calculates the magnetic moment around them. As soon as the vehicles equipped with this algorithm get close enough to moving obstacles, the magnetic moment keeps repulsing them away from the obstacles. Therefore, the combination of the static flow field and the dynamic dipole field enables autonomous vehicles to follow the pre-calculated paths and avoid unforeseen obstacles. Nevertheless, there is no research that verifies the algorithms within the context of autonomous vehicles, which is crucial before deploying them in concrete systems.

Modeling and Exhaustive Verification of the Algorithms. In this thesis, we encode the algorithms in the TA model of the embedded control software of the AWL and verify them in a discrete environment model by exhaustive model checking. The contribution is three-fold.

First, we model the functionalities of processes in the embedded control software by mapping the elements in the activity diagrams into the components of TA models. As depicted in Figure 5.2, the decision nodes, action nodes, and connections in Figure 5.2(a) are mapped to the locations, edges, and functions in the TA model in Figure 5.2(b). In addition, A* algorithm is programmed as a C-code function on the edge from location *Init* to *Configuration* in Figure 5.2(b). Second, we formalize the natural-language written requirements, such as *"The AWL must go to the primary crusher from the stone pile within*



(a) Activity diagram describing the function of a process



(b) TA of the process

Figure 5.2: Modeling the TA of a process from its activity diagram

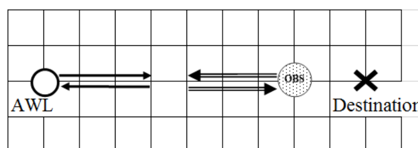


Figure 5.3: A livelock scenario

5 minutes", into TCTL queries so that they can be verified. For instance, the corresponding TCTL query of the requirement above-mentioned is formed as following:

```
(currentPos==pile and goal==crusher) −− >
(currentPos==pile and goal==pile and gClock <= 5 × 60)
```

Last but not least, by conducting exhaustive verification, we unveil a live-lock situation where two vehicles encounter each other on the same line but move to opposite directions forever, as it is shown in Figure 5.3. The reason that causes this undesired behavior is the limitation of the dipole flow field algorithm for collision avoidance, that is, the directions of the repulsive force generated by moving objects in this scenario are opposite to the directions of their velocities, and the attractive force always follows the planned path and pointing to

the destination. When they are getting close, the repulsive force pushes them away from each other. As their distance increases, the repulsive force becomes weaker and weaker until it is less than the attractive force, which is in the same line but towards the opposite direction of the repulsive force, and thus the autonomous vehicle and moving object are drawn by their destinations and move towards each other again. This leads to a forever live-lock situation.

By model checking, we are able to capture the phenomenon, hence providing valuable and our discovery provides valuable feedback to the algorithm's developers for further improvement. This work is reported in **Paper A** [7], which is also included as **Chapter 8** in this thesis.

Statistical Verification of the Algorithms. Collision avoidance relies strongly on the vehicles' dynamics and kinematics strongly, hence a discrete model does not suffice, as it cannot capture the continuous vehicle behavior. Such difficulties in adopting formal verification in realistic industrial scenarios have risen a wide interest in the academia [30, 31]. In this thesis, we apply hybrid automata and statistical model checking for modeling and verification of the collision-avoidance algorithm in a continuous environment model, where unforeseen moving obstacles are considered. The hybrid model is verified in UPPAAL SMC [16] and the results show that even in a dynamic environment with unforeseen moving obstacles that appear unexpectedly, the autonomous vehicle model is capable of avoiding the obstacles with a probability higher than 0.9. A more comprehensive description of the evaluation of the approach on industrial use cases is provided in Section 5.5. The simulation and verification results demonstrate that the dipole flow field algorithm can generate reasonable motions for the vehicles to avoid most of the moving obstacles. However, it may force the vehicles to move in cycles rather than stopping or detouring, to avoid collisions. These observations provide valuable feedback to the developers of the algorithm. Details of our contribution are reported in our **Paper B** [8] (included as **Chapter 9** in this thesis).

5.3 Scalable Synthesis of Collision-Free Mission Plans Via Model Checking and Reinforcement Learning

Mission planning includes path planning and task scheduling. Classic path-planning algorithms provide means of calculating static paths between two po-

sitions in the environment. Let us assume that the requirement of the mission plans of some autonomous wheel loaders is of the form: *"Dig stones at the stone pile. Carry and unload them into a primary crusher 500 meters away. Avoid static obstacles and keep repeating these tasks until the stone pile is empty or the vehicle needs to charge. Accomplish the job within 1 hour."* In such a case, one needs to employ path-planning algorithms and formal verification techniques in order to synthesize a mission plan that fulfills the complex requirement.

To utilize formal verification techniques on industrial systems, one has to build formal models of their behavior [32, 33]. This applies also to autonomous vehicles, which is our focus. An autonomous vehicle can be considered as an autonomous agent that is situated within an environment, can sense the environment and act on it, over time, in pursuit of its own goals [34]. In the work of solving the mission-planning problem of autonomous agents, whose movement and tasks are simply abstracted as time duration without considering any real-time feedback from the environment. Therefore, autonomous agents can be considered automated agents at this level of abstraction and defined as follows: *An automated agent is a system that receives instructions from its mission plan and executes its instructions with no human control and no interaction with its environment*. There are many definitions of automated agents in different fields of research [34]. In this thesis, we assume the definition above and formalize an automated agent as follows:

Definition (Automated Agent). *An automated agent (AA) is defined as a tuple:*

$$AA \triangleq < \mathcal{S}, \mathcal{M}, \mathcal{T} >, \tag{5.1}$$

where:

- $\mathcal{S}$ *is the speed of the moving agent,*
- $\mathcal{M}$ *is a set of motion primitives that make the agent move and execute tasks,*
- $\mathcal{T}$ *is a set of tasks that the agent has to accomplish.* □

Similarly, to lay out the foundation for generating formal models, we first contribute with defining the relevant concepts, such as autonomous agents, movement, and tasks. For example, the task execution is defined as a follows:

Definition (Task Execution). *For an automated agent* $(\mathcal{S}, \mathcal{M}, \mathcal{T})$*, the execution of tasks in* $\mathcal{T}$ *is defined as a timed automaton in a restricted form:*

$$Taa \triangleq (N, l_0, x_e, A_e, V_e, E_e, I_e, M_e) \tag{5.2}$$

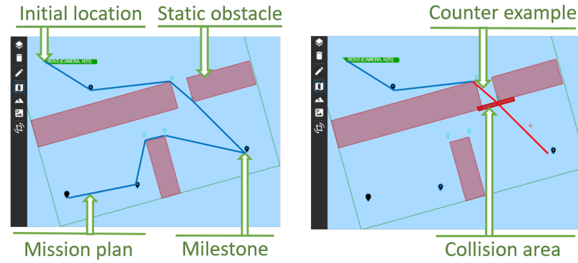
where,

- *N is a set of locations representing the tasks in $\mathcal{T}$,*
- *$l_0 \in N$ is the initial location representing the no-op task T_0,*
- *x_e is a clock that is reset whenever a task finishes,*
- *$A_e = \{move,\ done_0, ..., done_n\} \cup \tau$ is a set of actions,*
- *V_e is a set of variables containing variables of all the tasks in $\mathcal{T}$, i.e., $V_e = \bigcup_{i=1}^{S} T_i.V$, $S = |\mathcal{T}|$,*
- *$E_e \subseteq l_0 \times A_e \times B_e(x_e, \mathcal{T}) \times 2^C \times 2^{\mathcal{T}} \times N$ is a set of edges connecting l_0 and $l \in N$ with a set of actions and guards, where $C = \{x_e\}$,*
- *$I_e : N \setminus l_0 \mapsto B_i(x_e)$ is a function assigning invariants to locations except l_0,*
- *$M_e : N \mapsto \mathcal{T}$ is a function assigning tasks to locations.* □

Based on these formal definitions, we design model-generation algorithms to automatically generate the formal models needed for mission planning. The algorithms are implemented in a tool called *TAMAA (Timed-Automata-based planner for Multiple Autonomous Agents)*, which is connected to an graphic user interface called *MMT (Mission Management Tool)* [35]. After generating the formal model as a network of TA, TAMAA invokes the tool UPPAAL to verify the model and generate execution traces that satisfy or violate the desired properties, which are further leveraged to generate mission plans or counter examples that violate a specific formalized property, deeming generating a mission plan infeasible.

The contribution of TAMAA is to facilitate the mission plan synthesis process, by automating the model generation and integrating the tool for formal verification (UPPAAL) and the GUI for visualizing mission plans (MMT) in one framework. This tool allows designers to enjoy the benefits of formal methods without the need of becoming experts in the framework's theoretical underpinnings. As shown in Figure 5.4(a), in a reasonable environment where at least one path that goes through all milestones and reaches the destination exists, TAMAA outputs an execution trace as a witness, the mission plan being depicted in MMT. In the opposite case, a counter example representing an invalid mission plan is shown in MMT, where a collision occurs at the highlighted obstacle (See Figure 5.4(b)).

We also investigate the similarities and differences between our mission-planning problem and the classic job-shop problem so that we can leverage existing studies [25] and promote our algorithms. Our algorithms to generate mission plans for multiple autonomous agents consider not only the constraints of task execution but also timing requirements. The algorithms, contained in



(a) A mission plan generated in a reasonable environment (b) A counter example generated in an unreasonable environment

Figure 5.4: Two screenshots of the MMT user interface

our proposed MCRL approach, combine a reinforcement learning algorithm, i.e., Q-learning, with model checking to handle the state-space explosion problem when facing multiple agents. Uncertainties in this problem include the times of task execution and movement, which are time intervals following a uniform distribution. This makes the problem very complex and impossible to handle using traditional methods. The difficulties motivate the use of a combination of techniques, namely exhaustive model checking and reinforcement learning, which has the potential of reducing the state space to be explored by UPPAAL. Model checking provides the rigor and high assurance level, by formally encoding and analyzing the system that uses the described algorithms, whereas reinforcement learning compensates the method with respect to scalability and the handling of uncertainties.

The formal definitions, model-generation algorithms, and the TAMAA tool are described in **Paper C** [12], which is also included as **Chapter 10** in this thesis. We apply model checking alone, the combined analysis approach, namely MCRL, and UPPAAL STRATEGO [36] on an industrial use case that includes a number of autonomous wheel loaders and trucks working in a construction site. By applying our original approach involving model checking alone, we find that if the number of vehicles increases. e.g., to 5, the tool, i.e., TAMAA, fails to generate a mission. UPPAAL STRATEGO manages up to 2 agents, when synthesizing strategies based on the model generated by TAMAA. Our improved approach MCRL is proposed and evaluated in **Paper D**, which is also included as **Chapter 11** in this thesis, with the promising result of a linear increase of the synthesis time of mission plans, with the number of agents. The result of this study is overviewed in Section 5.5

5.4 Formal Modeling and Verification of the Embedded Control System and Dynamics of Autonomous Vehicles

Once mission plans are synthesized, we want to verify if they are correct (that is, meet the requirements) in the context of the embedded control software of the autonomous vehicles. To apply formal verification on a realistic model of autonomous vehicles, hence increasing the assurance via convincing verification results, we conduct the verification in a continuous and dynamic environment model and verify various requirements, like functional, precedence, and timing ones. For instance, the requirements of autonomous wheel loaders (AWL) are elaborated as following:

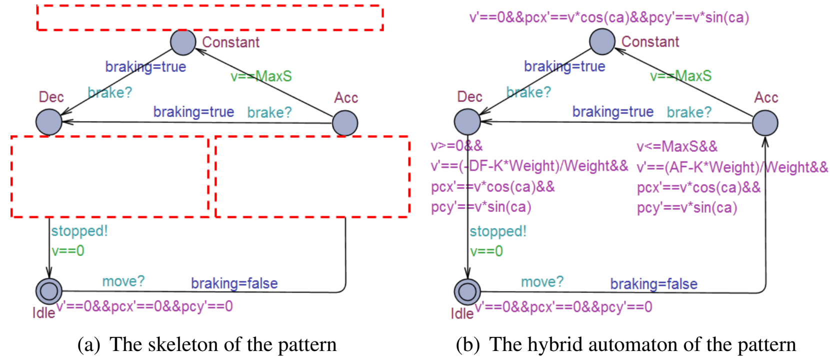
1) **Initial path computation**: during initialization, the autonomous vehicle must compute an initial path to the destination, which must avoid all the identified static obstacles;

2) **Obstacle avoidance and path recalculation**: the autonomous vehicle must avoid static and dynamic objects around it in due time before returning to the initial path;

3) **End-to-end deadline**: To guarantee a certain productivity, the autonomous vehicle must reach the destination within 30 minutes after having completed the entire mission.

To achieve this, we adopt hybrid-automata-based patterns to model the linear motion and rotation of the vehicles and statistical model checking in UPPAAL SMC [16] to verify the model against these requirements. Hybrid automata (HA) formalism is usually used to model systems in which digital computational processes interact with analog physical processes [17]. Based on Newtonian laws of motion, the HA of the linear motion of autonomous vehicles are modeled as depicted in Figure 5.5. In Figure 5.5(b), the derivatives of velocity, as well as the positions on x and y axis, are described by ODE, which are replaceable as shown in the pattern skeleton of Figure 5.5(a). Similarly, the rotation of vehicles is also modeled as well as the embedded control software.

As UPPAAL SMC does not support hierarchical or recursive modeling, the model tends to be extremely complex. All the elements of the systems at different levels, e.g., units, processes, threads, and functions, are constructed at the same level of the model. Therefore, we propose a **pattern-based method** for rapidly constructing complex models by reusing the common components



(a) The skeleton of the pattern (b) The hybrid automaton of the pattern

Figure 5.5: The pattern of the linear motion component of the autonomous vehicle

as shown in Figure 5.5. This approach facilities especially the modeling of the embedded control software. We refer to our **Paper B** [8], which is also included as **Chapter 9** in this thesis, for details and the evaluation on the use case of autonomous wheel loaders. The result proves that the method is capable of providing statistical verification of mission plan execution, considering the vehicles' movement in a dynamic environment.

5.5 Validating Our Solution on an Industrial Use Case: Autonomous Wheel Loaders

To validate the usability and scalability of our proposed approaches, we apply them on an industrial use case: an autonomous wheel loader (AWL), proposed by Volvo Construction Equipment, Sweden. The exhaustive verification of the AWL equipped with the A* algorithm for path planning, and dipole flow field algorithm for collision avoidance is reported in **Paper A** [7], which is also included as **Chapter 8**. Table 5.1 presents the TCTL queries and the verification results, from which we can conclude that the timed-automata-based method is able to provide a means of verification for the AWL within reasonable time, provided that the system complies with a set of assumptions, such as discretized environment and vehicle motions.

To inspect the continuous movement of the AWL and verify the system in a more realistic environment model, we simulate the hybrid-automata model created by using the pattern-based modeling method, by executing the simu-

Table 5.1: Verification queries and results of the AWL model equipped with A* and dipole flow field algorithms

Requirement	Query	Result	States explored	Time (ms)
Initial path computation	Q1.0: $E <>$ mainTask.Wait	Pass	2	110
	Q1.1: $A <>$ mainTask.Wait imply lenOfPathStack > 0	Pass	8780	484
	Q1.2: $E <>$ currentPosition == pile and destination == crusher	Pass	1	0
	Q1.3: (currentPosition == pile and destination == crusher) $-- >$ currentPosition == crusher	Pass	14191	1125
	Q1.4: $E <>$ currentPosition == crusher and destination == pile	Pass	2339	297
	Q1.5: (currentPosition == crusher and destination == pile) $-- >$ currentPosition == pile	Pass	14204	782
	Q1.6: $A[]$ forall(i:int[0,9]) currentPosition != staticObstacle[i]	Pass	8780	485
Obstacle avoidance	Q2.0: $A[]$ currentPosition != currentObstacle	Pass	125941	6297
	Q1.3: (currentPosition == pile and destination == crusher) $-- >$ currentPosition == crusher	Pass	227646	13969
	Q1.4: $E <>$ currentPosition == crusher and destination == pile	Pass	2678	375
	Q1.5: (currentPosition == crusher and destination == pile) $-- >$ currentPosition == pile	Pass	192406	10656
Mode switch: error A	Q3.1: $E <>$ errorStart == true	Pass	30	234
	Q3.2: error_start==true $-- >$ (SYSTEM_ERROR==true and reaction_time$<=$20)	Pass	91	250
Mode switch: error B	Q3.1: $E <>$ errorStart == true	Pass	29	234
	Q3.2: error_start==true $-- >$ (SYSTEM_ERROR==true and reaction_time$<=$15)	Pass	320	266
End-to-end deadline	Q4.0: (currentPosition==pile and destination==crusher) $-- >$ (currentPosition==pile and destination==pile and gClock $<=$ 2200)	Pass	590326	36641

lation query (5.3) in UPPAAL SMC, to obtain the changing coordinates of the autonomous vehicle:

```
simulate 1[<=110] {pcx,pcy}
```
(5.3)

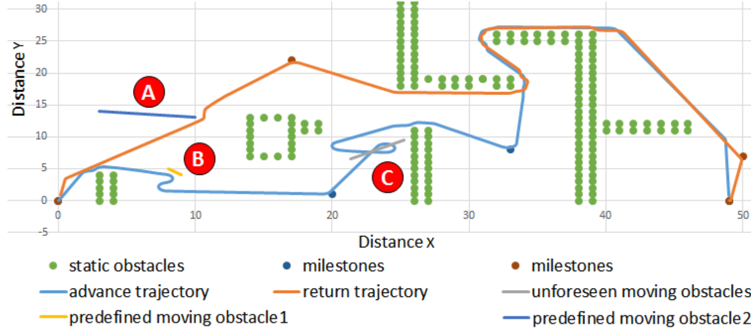


Figure 5.6: The trajectory of the AWL in a map with three moving obstacles

The trajectory is shown in Figure 5.6, where “A” and “B” are two predefined moving obstacles and “C” is a dynamically-generated obstacle that moves “recklessly” towards the AWL. The AWL is capable of avoiding the predefined moving obstacles “A” and “B” and the trajectory does not deviate much from the initial path plan, whereas when encountering the unforeseen moving obstacle “C”, the AWL has to turn around to avoid collision. Besides the simulation, we also conduct statistical model checking on the hybrid-automata model against queries like the following:

$$\texttt{Pr[<=70](<> arrived \&\& counter<=60)} \tag{5.4}$$

$$\texttt{Pr[<=110]([] followedPath),} \tag{5.5}$$

where in query (5.4), `arrived` is a boolean variable denoting if the AWL has arrived at destination, and `counter` is a clock used to encode the associated timing constraint. In query (5.5), the boolean variable `followedPath` models the fact that the AWL has reached the destination and has returned to the start by visiting all the required milestones orderly. The probability interval of satisfying these queries is [0.902606, 1] with 95% confidence obtained based on 36 runs. The verification results demonstrate that the path-planning algorithms work correctly, whereas the dipole flow field algorithm needs to be improved to cope with some special scenarios. This work is documented in **Paper B** [8] (included as **Chapter 9**).

The algorithms and tool for mission planning, namely TAMAA and MCRL, are evaluated on the same industrial use case to demonstrate the usefulness and scalability of proposed verification solutions. The experiment is conducted on a machine running an Intel Core i5 processor with 16 GB of RAM and a

Table 5.2: Scalability evaluation results of TAMAA with different number of milestones and tasks and 1 vehicle.

Query	Numer of Milstones	Numer of Tasks	Numer of Explored States	Time
Reachability	30	30	20,363	0.2 s
	60	60	157,033	2.2 s
	100	100	712,721	14 s
Invariance	30	30	41,193	0.3 s
	60	60	317,703	4.5 s
	100	100	1,429,903	29 s

Table 5.3: Scalability evaluation results of TAMAA with different number of vehicles running 3 tasks among 3 milestones.

Query	Numer of Vehicles	Numer of Explored States	Time
Reachability	2	1,661	0.01 s
	3	159,632	2.0 s
	4	2,058,132	20160 s
	5	Out of Memory	Out of Memory
Invariance	2	3,533	0.03 s
	3	344,701	4.0 s
	4	Out of Memory	Out of Memory

64-bit Windows OS. As shown in Table 5.2, when the numbers of milestones and tasks increase, the computation time of TAMAA increases acceptably. In contrast, when the number of vehicles increases to 5, TAMAA encounters state-space explosion problem, and fails to terminate with a result, as shown in Table 5.3. This work is reported in **Paper C** [12], which is also included as **Chapter 10** in this thesis.

Our new approach of scalable mission planning for multiple vehicles, namely MCRL, is developed based on TAMAA and reported in **Paper D** [24] (**Chapter 11** in this thesis). This approach utilizes reinforcement learning to restrict the behavior of agents, and thus the state-space of the model is contained. The new agent model obtained from MCRL is able to find the desired states faster and cope with uncertain movement time and task execution time. Experiments aiming at comparing this new approach with the original TAMAA, and another benchmark, namely UPPAAL STRATEGO [36] is also conducted in this study. Figure 5.7 shows the computation time of running the three methods to syn-

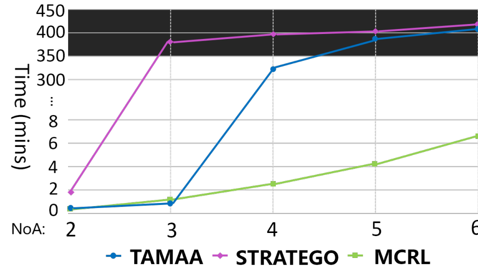


Figure 5.7: The time consumption of different methods running different number of agents

Table 5.4: Contribution of included papers with respect to research goals

	SG 1	SG 2	SG 3	SG 4	SG 5
Paper A		X			X
Paper B	X			X	X
Paper C			X		X
Paper D			X		X

thesize mission plans for 2 to 6 autonomous agents in the same scenario. The values in the black region reflect the times when the physical memory assigned to the process is exhausted, due to the model's state-space explosion. The result of this study shows that the computation time when employing MCRL increases linearly with the number of agents, whereas TAMAA and UPPAAL STRATEGO show an exponential increase of the computation time, hence they fail to output a result, for 5 agents, and 3 agents problem size, respectively.

5.6 Research Goals Revisited

In this section, we present the technical contributions of this thesis and the relationship between the included papers and the research goals. Each research goal is addressed by one or more papers, as illustrated in Table 5.4.

- **Paper B** [8] proposes a two-layer framework for the formal modeling and verification of autonomous vehicles, such that designers can utilize formal methods to analyze and design autonomous vehicles via a systematic approach that is founded on rigorous design elements based on

formal models. This contribution addresses **Subgoal 1** (SG1): *Provide a means that decouples the design of mission planning from the vehicle's autonomous operation in a continuous dynamic environment, supported by model checking techniques*.

- **Paper A** [7] applies exhaustive model checking to verify the design of an autonomous wheel loader prototype, equipped with path-planning and collision-avoidance algorithms, such as A* and dipole flow field, respectively. This contribution addresses **Subgoal 2** (SG2): *Ensure the correctness of path-planning and collision-avoidance algorithms, within the context of autonomous wheel loaders*.

- **Paper C** [12] proposes the TAMAA approach that provides a rigorous theoretical foundation of mission-plan synthesis, based on formal models, and a graphical user interface for environment and vehicle configuration. **Paper D** [24] improves TAMAA by combining it with reinforcement learning. The new approach can alleviate the state-space explosion problem and manage to handle cases with more than 5 agents in different scenarios. These contributions address **Subgoal 3** (SG3): *Provide scalable synthesis of collision-free static mission plans guaranteed to satisfy given temporal requirements among tasks*.

- **Paper B** [8] proposes a pattern-based approach for the formal modeling of the behavior of an AWL by using hybrid automata, and applies statistical model checking to verify the resulting model. These contributions address **Subgoal 4** (SG4): *Ensure that the model execution of the vehicle's movement in a dynamic environment fulfills the specified functional, precedence, and timing requirements*.

- By applying our proposed methods to the industrial use case of an AWL prototype, **papers A** [7], **B** [8], **C** [12] and **D** [24] address **Subgoal 5** (SG5): *Assess the applicability of the proposed synthesis and verification methods on an industrial use case*.

These contributions together provide a solution for designing the embedded control software of autonomous vehicles and ensuring its correctness, with respect to mission planning, as well as path following and collision avoidance functions, which addresses our overall research goal (see Section 3.2).

Chapter 6

Related Work

In this chapter, we present some of the related work in mission planning of autonomous vehicles, and verification of autonomous vehicles in a dynamic, continuous environment model. These previous studies pave the way towards facilitating the mission planning and verification of autonomous agents. Our work is also inspired by some of the related work mentioned below. However, the fact that these related studies either consider only one aspect of the problem, i.e., discrete mission planning or verification of hybrid models, or fail to provide a scalable solution for multiple agents in one framework motivates us to extend the study and fill the research.

6.1 Mission Planning for Autonomous Agents

In recent decades, there has been a growing interest in formal modeling and verification of autonomous systems, especially for mission planning problems with complex goals. Belta et al. [37] present a hierarchical structure, and based on a three-level process they propose a method for the verification of mobile robots using Linear Temporal Logic (LTL). This is evaluated in several case studies [38, 39]. Bhatia et al. [40, 3] propose a multi-layered synergistic approach for solving motion planning problems for mobile robots involving temporal goals. This approach addresses two key issues: the construction of the discrete abstraction of the robots and its efficient exploration in the high-level layer. Dimarogonas et al. [5, 41] propose their method for motion planning of multiple-agent systems using various temporal logics. Saddem et al.

[42] use UPPAAL and Computation Tree Logic (CTL) to verify reachability properties of autonomous functionalities, including path finding. The authors propose an environment decomposition method to reduce the memory requirement and execution time of model checking. Koo et al. [4] propose a framework for the coordination of a network of mobile robots with respect to formal requirement specifications in temporal logics, in which hybrid automata and Cadence's SMV model checker are used.

As different from these studies, our approach is focusing on integrating a state-of-the-art path-planning algorithm with temporal logic, to leverage the heuristics and efficiency of the former and the rigorousness and expressiveness of the latter. In addition, our approach combines the model checking technique with reinforcement learning to alleviate the state-space explosion problem, hence being able to handle larger problem sizes than those handled by the mentioned related approaches (e.g., involving a larger number of autonomous agents, or milestones and tasks).

The combination of formal methods and learning algorithms is a recent trend that attracts a large body of research work. Li et al. [43] utilize the expressiveness of formal specification languages to capture complex requirements of robotic systems, and construct reward functions of reinforcement learning so that they become interpretable. Bouton et al. [44] propose a generic approach to enforce probabilistic guarantees on agents trained by reinforcement learning. Mason et al. [45] present an assured reinforcement learning algorithm, using abstract Markov decision processes, and probabilistic model checking to establish abstract policies for autonomous agents that are formally verified. UPPAAL STRATEGO as a new branch of UPPAAL is designed by David et al. [36], and adopts reinforcement learning algorithms to refine the synthesized strategies for winning priced timed games.

In comparison to the above work, our approach focuses on using reinforcement learning as a way of taming the scalability of exhaustive model checking, for mission-plan synthesis of multi-agents, so that the state-space explosion is alleviated. The model-checking technique compensates the reinforcement learning algorithms by providing formal guarantees of satisfying requirements that are not expressed in the reward functions of the learning algorithms.

We also integrate the mission plan synthesis method with a GUI, namely MMT [35], which allows the easy configuration of the system and its environment. We aslo apply the approach on an industrial case involving autonomous wheel loaders, to demonstrate the applicability and to some extent also the scalability of this approach in realistic scenarios. Instead of using LTL, as in some of the related work [37], for requirements specification, we explore the

use of Timed Computation Tree Logic (TCTL) for expressing different types of requirements like requirements with timing constraints.

6.2 Verification of Autonomous Agents

Automata-based methods [38, 46, 47, 48] have been used for path or motion planning fow a while now. Different from our work, which utilizes and verifies path-planning algorithms, these studies aim to solve the vehicle-routing problem by exploring the environment model using model checking. In the related papers, the authors study agents that carry out autonomous tasks like searching for an object, avoiding an obstacle, and missions sequencing. However, uncertainties, like unforeseen obstacles, which are hard to predict, have not been considered.

Runtime verification that monitors the behavior of autonomous systems addresses the above-mentioned shortage to some extent [49, 50, 51, 52]. This technique extracts information from a running system, based on which the behavior of the system is verified. The runtime overhead caused by the monitor is the most common problem introduced by this method.

The agent-based paradigm is another widely studied approach for the design and analysis of autonomous systems [53, 54, 54, 55, 56]. Since the predominant form of rational agents architecture is that provided by the Beliefs, Desires, and Intentions approach, these studies aim to translate the agent-based language to a formal notation, in order to be able to verify the behavior of the agents. However, this method usually does not consider the continuous dynamics of the vehicle, which we model and analyze in the dynamic layer of our two-layer framework.

There are also some studies providing frameworks for verification of autonomous vehicles or robots. Sirigineedi et al. [57], capture the behavior of an unmanned aerial vehicle performing cooperative search mission into a Kripke model to verify it against the temporal properties expressed in CTL. The model used in the paper contains a decision-making layer and a path-planing layer. Quilbeuf et al. [58] propose a generic method based on Statistical Model Checking (SMC) to evaluate complex automotive-oriented systems. They use specifically defined Key Performance Indicators (KPIs) as temporal properties and evaluate the probability of the systems to meet the KPIs. Desai et al. [59] propose an approach that combines model checking with runtime verification to bridge the gap between software verification (discrete) and the actual execution of the software on a real robotic platform in the physical world. However,

it does not model and verify the vehicles' behaviors that are governed by the generated paths and their dynamics in a continuous environment model.

Chapter 7

Conclusions and Future Work

In this thesis, we first propose a two-layer framework for the mission planning and verification of autonomous vehicles. Following the well-known principle of separation of concerns, the framework decouples the discrete mission planning from the verification of concrete execution and collision avoidance in continuous environments. The framework consists of a static layer that is responsible for mission planning, and adopts a combination of model checking and reinforcement learning, and the dynamic layer that is intended for the verification of the mission plans' execution, and considers the dynamics and kinematics of the autonomous vehicles and unforeseen moving obstacles.

To facilitate mission planning, we support the framework by a tool called TAMAA (Timed-Automata-based planner for Multiple Autonomous Agents), which implements our model-generation algorithms and connects to UPPAAL and a GUI for mission management, called MMT. The TAMAA approach provides rigorous formal definitions of important concepts in the mission-planning problem, e.g., agent movement and task execution. These definitions establish the foundation for the automatic model-generation algorithms that serve as another contribution of this thesis. TAMAA implements these algorithms and integrates MMT, so that designers need not use or be experts in the underlying formal notations and methods, instead they can focus solely on the environment configuration and requirements specification. Nevertheless, mission plans are guaranteed to be correct thanks to the formal modeling and verification techniques that are employed in TAMAA.

To improve TAMAA's ability of handling multiple agents, we combine the model checking technique with reinforcement learning (MCRL), and conduct a series of experiments to compare our new approach with the original TAMAA, and UPPAAL STRATEGO, respectively. The experimental results show that MCRL is able to handle more than 5 agents in different scenarios, whereas the original TAMAA can only deal with a maximum of 4 agents, and UPPAAL STRATEGO with maximum 2. This novel approach alleviates the widely-known state-space-explosion problem of model checking, in the mission-planning domain for multiple autonomous agents. Moreover, it also provides a means of synthesizing mission plans with guaranteed correctness, which bests original reinforcement learning algorithms.

For the dynamic layer of the framework, we propose a model in the framework of hybrid automata, to describe the discrete state transition of the systems, as well as kinematics of autonomous vehicles and unforeseen obstacles. As the embedded control software is complex, we propose a pattern-based modeling method to facilitate the modeling process and enable reuse. We have demonstrated the feasibility of the approach, by building models in UPPAAL SMC and conducting a series of statistical analysis of a real-world industrial properties, namely the autonomous wheel loader use case, provided by Volvo Construction Equipment, in Sweden. This use case serves as our main source of research problems, and motivates the progress of our work, by providing problem scenarios and valuable materials, such as system requirements and initial designs. The result of applying our framework of design and verification on this use case demonstrates the applicability and, to some extent, scalability of our method and tool.

7.1 Limitations

Although promising, the TAMAA approach, which focuses on the static layer of the framework, and the pattern-based modeling method of the dynamic layer have not been integrated as one complete framework, in the sense that they do not communicate in real-time and automatically. This shortage limits the autonomous agents' ability of re-planning in case of meeting anomalies, i.e., temporal obstacles, or forbidden areas. As the environment is uncertain, a large inaccessible area or obstacle may appear while the agent is traveling, such as a big hole on the ground in a construction site, or an area of wet floor in an indoor environment where robotic wheel chairs are helping the disabled to move. In these scenarios, collision-avoidance algorithms may cause the agents to deviate

too much from the original path, and a re-planning is strongly needed to achieve the long-term goals, such as guaranteeing productivity, or reaching restrooms in time.

Secondly, the MCRL approach initiates a direction of solving the state-space-explosion problem of model checking, and the initial experiment shows a promising result when assuming large numbers of agents. However, the algorithm still separates the iteration of simulation and reinforcement learning in different phases, namely the data gathering phase and model training phase. This limits the effect of learning while the state space of the model is being explored during the simulation. In other words, if in each round or even step of the simulation reinforcement learning could take place, the entire exploration steps should be much decreased when checking reachability properties. This heuristic approach of state-space exploration can benefit the reduction of the effort of building up the state space of a formal model, and reaching the desired states with less meaningless exploration.

7.2 Future Work

The future work has several possible directions. One is to integrate the two layers of the framework so that they communicate in a real-time manner, and the mission planning and verification are both optimized in this way. In theory, this work needs effective interaction between the discrete and continuous areas of the system. In practice, it requires a competent and elegant design mix of hybrid automata, timed automata, and communication between them. The two-layer framework provides a separation of concerns of system design, while the integration of the two layers offers a flexible and realistic means of verification. Altogether they can bring the advantages of adopting formal methods during the design of a real-world system (like the AWL) closer to industrial end-users' attention.

Another direction concerns improving the MCRL approach by embedding the reinforcement learning into the state-space exploration of the model when running verification. This can be achieved by leveraging the calling of external functions in UPPAAL STRATEGO, or by implementing a customized model checker by leveraging existing libraries and frameworks, such as Plasma [60], and Storm [61].

Investigating new methods to solve the outstanding problems that cannot be addressed by the current methods can be another interesting direction. A systematic study will be finished, to provide an overview of the research area of

applying formal methods in the design of autonomous vehicles. Problems such as the design of heterogeneous autonomous vehicles that cooperate in the sites, and defining “a deployment model” with respect to the possible heterogeneous architecture that the AWL use case could be implemented on probably need new methods to solve.

Bibliography

[1] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[2] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381, 2009.

[3] Amit Bhatia, Matthew R Maly, Lydia E Kavraki, and Moshe Y Vardi. Motion planning with complex goals. *IEEE Robotics & Automation Magazine*, 18(3), 2011.

[4] T John Koo, Rongqing Li, Michael M Quottrup, Charles A Clifton, Roozbeh Izadi-Zamanabadi, and Thomas Bak. A framework for multi-robot motion planning from temporal logic specifications. *Science China Information Sciences*, pages 1–18, 2012.

[5] Fernando S Barbosa, Lars Lindemann, Dimos V Dimarogonas, and Jana Tumova. Integrated motion planning and control under metric interval temporal logic specifications. In *2019 18th European Control Conference (ECC)*. IEEE, 2019.

[6] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[7] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Formal verification of an autonomous wheel loader by model checking. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, pages 74–83. ACM, 2018.

[8] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Towards a two-layer framework for verifying autonomous vehicles. In *NASA Formal Methods Symposium*, pages 186–203. Springer, 2019.

[9] Steve Rabin. Game programming gems, chapter a* aesthetic optimizations. *Charles River Media*, 2000.

[10] Lan Anh Trinh, Mikael Ekström, and Baran Cürüklü. Toward shared working space of human and robotic agents through dipole flow field for dependable path planning. *Frontiers in neurorobotics*, 12, 2018.

[11] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, 3098:87–124, 2004.

[12] Rong Gu, Eduard Paul Enoiu, and Cristina Seceleanu. Tamaa: Uppaal-based mission planning for autonomous agents. In *The 35th ACM/SIGAPP Symposium On Applied Computing*, April 2020.

[13] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

[14] Rajeev Alur and David Dill. The theory of timed automata. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 45–73. Springer, 1991.

[15] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[16] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. *arXiv preprint arXiv:1208.3856*, 2012.

[17] Thomas A Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[18] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[19] Paul E Black. Manhattan distance. *Dictionary of Algorithms and Data Structures*, 18:2012, 2006.

[20] Luis Valbuena and Herbert G Tanner. Hybrid potential field based control of differential drive mobile robots. *Journal of intelligent & robotic systems*, 68(3-4):307–322, 2012.

[21] Yoav Golan, Shmil Edelman, Amir Shapiro, and Elon Rimon. Online robot navigation using continuously updated artificial temperature gradients. *IEEE Robotics and Automation Letters*, 2(3):1280–1287, 2017.

[22] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[23] Mykel J Kochenderfer. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[24] Rong Gu, Eduard Paul Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Combining model checking and reinforcement learning for scalable mission planning of autonomous agents. Mälardalen Real-Time Research Centre, Mälardalen University.

[25] Yasmina Abdeddaı, Eugene Asarin, Oded Maler, et al. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

[26] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[27] Marvin V Zelkowitz and Dolores Wallace. Experimental validation in software engineering. *Information and Software Technology*, 39(11):735–743, 1997.

[28] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE software*, 12(4):52–62, 1995.

[29] Hilary J Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research methods in computing: what are they, and how should we teach them? *ACM SIGCSE Bulletin*, 38(4):96–114, 2006.

[30] Mattias Nyberg, Dilian Gurov, Christian Lidström, Andreas Rasmusson, and Jonas Westman. Formal verification in automotive industry: Enablers and obstacles. In *International Symposium on Leveraging Applications of Formal Methods*, pages 139–158. Springer, 2018.

[31] Michael Felderer, Dilian Gurov, Marieke Huisman, Björn Lisper, and Rupert Schlick. Formal methods in industrial practice-bridging the gap (track summary). In *International Symposium on Leveraging Applications of Formal Methods*, pages 77–81. Springer, 2018.

[32] Marta Olszewska, Sergey Ostroumov, and Marina Waldén. Using scrum to develop a formal model–an experience report. In *International Conference on Product-Focused Software Process Improvement*, pages 621–626. Springer, 2016.

[33] Marieke Huisman, Dilian Gurov, and Alexander Malkis. Formal methods: From academia to industrial practice. a travel guide. *arXiv preprint arXiv:2002.07279*, 2020.

[34] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer, 1996.

[35] B. Ekström M. Ameri E. A., Çürüklü. Planning and supervising autonomous underwater vehicles through the mission management tool. Submitted to Oceans Conference and Exposition, IEEE, 2020.

[36] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal stratego. In *TACAS*. Springer, 2015.

[37] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George J Pappas. Symbolic planning and control of robot motion [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):61–70, 2007.

[38] Stephen L Smith, Jana Tumova, Calin Belta, and Daniela Rus. Optimal path planning for surveillance with temporal-logic constraints. *International Journal of Robotics Research*, 30(14):1695–1708, 2011.

[39] Alphan Ulusoy, Stephen L Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. Optimality and robustness in multi-robot path planning with temporal logic constraints. *International Journal of Robotics Research*, 32(8):889–911, 2013.

[40] Amit Bhatia, Lydia E Kavraki, and Moshe Y Vardi. Sampling-based motion planning with temporal goals. In *International Conference on Robotics and Automation*, pages 2689–2696. IEEE, 2010.

[41] Alexandros Nikou, Dimitris Boskos, Jana Tumova, and Dimos V Dimarogonas. On the timed temporal logic planning of coupled multi-agent systems. *Automatica*, 97:339–345, 2018.

[42] Rim Saddem, Olivier Naud, Karen Godary Dejean, and Didier Crestani. Decomposing the model-checking of mobile robotics actions on a grid. *IFAC-PapersOnLine*, 50(1):11156–11162, 2017.

[43] Xiao Li, Zachary Serlin, Guang Yang, and Calin Belta. A formal methods approach to interpretable reinforcement learning for robotic planning. *Science Robotics*, 4(37), 2019.

[44] Maxime Bouton, Jesper Karlsson, Alireza Nakhaei, Kikuo Fujimura, Mykel J Kochenderfer, and Jana Tumova. Reinforcement learning with probabilistic guarantees for autonomous driving. *arXiv preprint arXiv:1904.07189*, 2019.

[45] George Rupert Mason, Radu Constantin Calinescu, Daniel Kudenko, and Alec Banks. Assured reinforcement learning with formally verified abstract policies. In *ICAART*, 2017.

[46] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2020–2025. IEEE, 2005.

[47] Marius Kloetzer and Cristian Mahulea. A petri net based approach for multi-robot path planning. *Discrete Event Dynamic Systems*, 24(4):417–445, 2014.

[48] Michael Melholt Quottrup, Thomas Bak, and RI Zamanabadi. Multi-robot planning: A timed automata approach. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4417–4422. IEEE, 2004.

[49] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[50] Erann Gat, Marc G Slack, David P Miller, and R James Firby. Path planning and execution monitoring for a planetary rover. In *Proceedings of*

the IEEE International Conference on Robotics and Automation, pages 20–25, 1990.

[51] Alex Lotz, Andreas Steck, and Christian Schlegel. Runtime monitoring of robotics software components: Increasing robustness of service robotic systems. In *Advanced Robotics (ICAR), 2011 15th International Conference on*, pages 285–290. IEEE, 2011.

[52] Chenxia Luo, Rui Wang, Yu Jiang, Kang Yang, Yong Guan, Xiaojuan Li, and Zhiping Shi. Runtime verification of robots collision avoidance case study. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 204–212. IEEE, 2018.

[53] Rafael H Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Autonomous agents and multi-agent systems*, 12(2):239–256, 2006.

[54] 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.

[55] Michael Fisher, Rafael H Bordini, Benjamin Hirsch, and Paolo Torroni. Computational logics and agents: a road map of current technologies and future trends. *Computational Intelligence*, 23(1):61–91, 2007.

[56] Michael Fisher, Louise Dennis, and Matt Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, 2013.

[57] Gopinadh Sirigineedi, Antonios Tsourdos, Brian A White, and Rafal Zbikowski. Modelling and verification of multiple uav mission using smv. *arXiv preprint arXiv:1003.0381*, 2010.

[58] Jean Quilbeuf, Mathieu Barbier, Lukas Rummelhard, Christian Laugier, Axel Legay, Blanche Baudouin, Thomas Genevois, Javier Ibañez-Guzmán, and Olivier Simonin. Statistical model checking applied on perception and decision-making systems for autonomous driving. 2018.

[59] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. Combining model checking and runtime verification for safe robotics. In *International Conference on Runtime Verification*, pages 172–189. Springer, 2017.

[60] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Plasma lab: a modular statistical model checking platform. In *International Symposium on Leveraging Applications of Formal Methods*, pages 77–93. Springer, 2016.

[61] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 592–600, Cham, 2017. Springer International Publishing.

II

Included Papers

Chapter 8

Paper A: Formal Verification of an Autonomous Wheel Loader by Model Checking

Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist.
Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE). ACM, 2018.

Abstract

In an attempt to increase productivity and the workers' safety, the construction industry is moving towards autonomous construction sites, where various construction machines operate without human intervention. In order to perform their tasks autonomously, the machines are equipped with different features, such as position localization, human and obstacle detection, collision avoidance, etc. Such systems are safety critical, and should operate autonomously with very high dependability (e.g., by meeting task deadlines, avoiding (fatal) accidents at all costs, etc.). An Autonomous Wheel Loader is a machine that transports materials within the construction site without a human in the cab. To check the dependability of the loader, in this paper we provide a timed automata description of the vehicle's control system, including the abstracted path planning and collision avoidance algorithms used to navigate the loader, and we model check the encoding in UPPAAL, against various functional, timing and safety requirements. The complex nature of the navigation algorithms makes the loader's abstract modeling and the verification very challenging. Our work shows that exhaustive verification techniques can be applied early in the development of autonomous systems, to enable finding potential design errors that would incur increased costs if discovered later.

8.1 Introduction

Industrial robots are used in modern manufacturing sites to automate repetitive tasks and reduce labor costs. Advances in self-driving vehicles have propelled similar developments in the construction industry, by the outset of autonomous construction equipment, which are heavy vehicles that operate without human intervention.

The environment where the autonomous construction equipment operates is hazardous, that is, dusty, with possibly harsh weather conditions, and populated with static and dynamic obstacles that need to be discovered and avoided by all means. These vehicles are designed to perform predefined tasks, and, unlike industrial robots, they operate in large construction sites, alongside other vehicles and humans. On the one hand, their environment is contained and controlled, thus their autonomy is bounded. On the other hand, being complex safety-critical systems, the autonomous construction equipment's dependability is crucial for ensuring safety and increased productivity, hence verifying formally an abstraction of the system's behavior could be highly beneficial. In this paper, we take upon such a task and formally model and verify an industrial prototype of an autonomous wheel loader against functional, timing, and safety requirements. The complexity of the system stems from the integrated intelligent algorithms, such as path planning, obstacle detection, and collision avoidance, etc. The crux of our work is the formalization of an abstraction of the vehicle's motions, control system, path-planning and collision-avoidance algorithms, such that resulting model is analyzable via exhaustive model checking. We use the timed automata (TA) [1] framework for modeling, and the UPPAAL [2] model checker for verification.

In comparison to related efforts of verifying autonomous vehicles [3, 4, 5, 6], our approach encodes the A* algorithm [7] for initial path planning, as well as the dipole flow field algorithm [8] used for avoiding static and dynamic obstacles, which are two algorithms that resolve many issues of implementing reliable collision avoidance effectively. Both algorithms are encoded as C functions in UPPAAL. To create the model of the machine's control system, we map the activity diagrams of components to TA representations. The system requirements, initially described in natural language, are formalized in Timed Computation Tree Logic (TCTL), as UPPAAL queries that the formal model needs to satisfy for any possible behavior. We show that under the mentioned abstractions, the exhaustive verification of the autonomous loader is possible, and we also discuss some identified issues of verifying a more faithful model.

This paper is organized as follows. In Section 8.2, we present the architec-

Figure 8.1: The AWL in its working environment

ture of the autonomous wheel loader, as well as its natural language requirements. Section 8.3 overviews the preliminaries, that is, timed automata and UPPAAL, as well as the A* algorithm for the loader's initial path planning, and the dipole flow field algorithm for collision avoidance. In Section 8.4, we show the TA model of the loader's control tasks and algorithms, the verification queries and model checking results. A short discussion and lessons learned are provided in Section 8.5, after which we compare to related work in Section 8.6. Finally, Section 8.7 concludes the paper.

8.2 Autonomous Wheel Loader: Architecture and Requirements

In this section, we introduce the Autonomous Wheel Loader (AWL), which is an industrial prototype and serves as our use case. The AWL is a heavy vehicle used in the construction site to transport materials (e.g., blasted rocks), which works independently, without any manual intervention. The AWL operates in a quarry (see Figure 8.1), where it transports rocks between a stone pile and a crusher. To be able to operate autonomously, the AWL is equipped with a path planning system that computes the initial path from the stone pile to crusher and back, which the AWL should follow. We assume that there are various obstacles in the quarry, such as humans, other machines, holes, signs, etc. Other functions like autonomous digging, unloading etc. are not considered here. To ensure safety, the AWL is equipped with a collision avoidance system that identifies nearby objects, and deviates from the planned path (i.e., changes the direction and possibly the speed of the AWL), if needed, to avoid collision. This mechanism should cope with different light conditions (from bright sunlight to complete darkness), possibly bad weather (heavy rain or snow), dust, etc. To ensure it perceives its surroundings accurately, the AWL has a set of

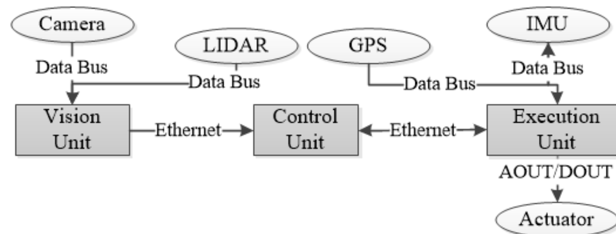


Figure 8.2: The architecture of the AWL's control system

sensors, including GPS and IMU (Inertial Measurement Unit) for localization, and LIDAR, radar and camera for obstacles capture and identification.

The architecture of the AWL's control system, presented in Figure 8.2, consists of three main units: the vision unit, the control unit, and the execution unit, which are connected via Ethernet. The roles of these units are as follows:

- The vision unit is connected to the LIDAR and camera, and is responsible for detecting obstacles within the vision range.
- The control unit collects data (e.g., position of the AWL, obstacles, system status, etc.) from other units, plans the path, schedules the tasks, and sends commands to the execution unit.
- The execution unit controls the actuators, the steering and the brakes, based on the commands received from the control unit. It also collects data from the GPS and IMU, and sends them to the control unit.

AWL's Functionality. The functionality of the system is implemented through a set of tasks that are assigned and executed on the three units respectively, as depicted in Figure 8.3.

The obstacle detection relies on the *Do Obstacle Task* in the Vision Unit. This task is responsible for: (i) acquiring data from the sensors (e.g., LIDAR, camera), and (ii) executing the recognition algorithms to determine the presence and the type of the obstacles (e.g., human, other machines, holes).

The Control Unit executes three parallel tasks, described below:

- *Read Position Task* that reads the loader's position from the Execution Unit,
- *Main Task* that is responsible for generating the initial path, analyzing the environment, and devising control strategies to avoid different obstacles,

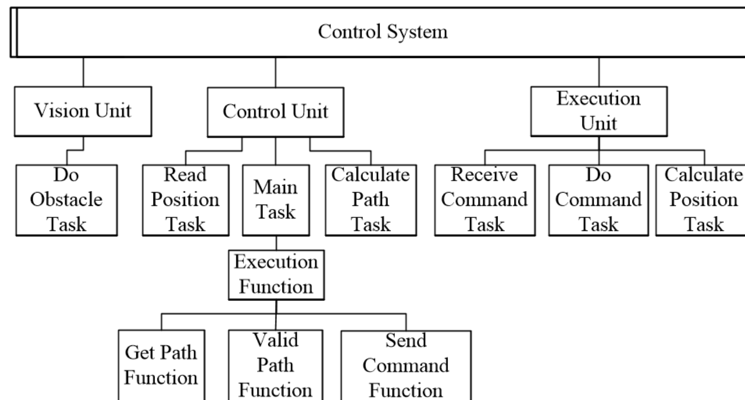


Figure 8.3: Task allocation in the control system

- *Calculate Path Task* that calculates a new path when the AWL encounters an obstacle and deviates from the initial path.

Three parallel tasks are assigned to the Execution Unit, namely *Receive Command Task*, *Do Command Task*, and *Calculate Position Task*. The tasks are responsible for getting commands from the Control Unit, executing the commands to move or brake the AWL, and calculating the position of the AWL and sending it to the Control Unit, respectively.

The communication among these tasks is asynchronous, that is, the tasks do not await response after they send out data. The tasks interact and cooperate with each other to accomplish specific missions of the control system, e.g., perceiving information from the environment, formulating an efficient (or close to optimal) path to avoid a dynamic obstacle, etc. Figure 8.4 depicts the partial interaction between tasks. *Main Task* takes one path segment of the initial path from *Path Stack 1*, which stores the initial path in the control unit. Next, it calls the *Valid Path Function* to check if the path segment leads to any collision. If the validation passes, the path segment is sent to *Receive Command Task* in the execution unit. Otherwise, the AWL might encounter an obstacle or malfunction, in which case *Calculate Path Task* will receive a new path request from the *Main Task*. Consequently, the corresponding algorithm employed for collision avoidance, called the dipole field algorithm [8], is executed in *Calculate Path Task* before a new path segment is sent to *Receive Command Task*, if it exists. If the calculation does not return any new path segment,

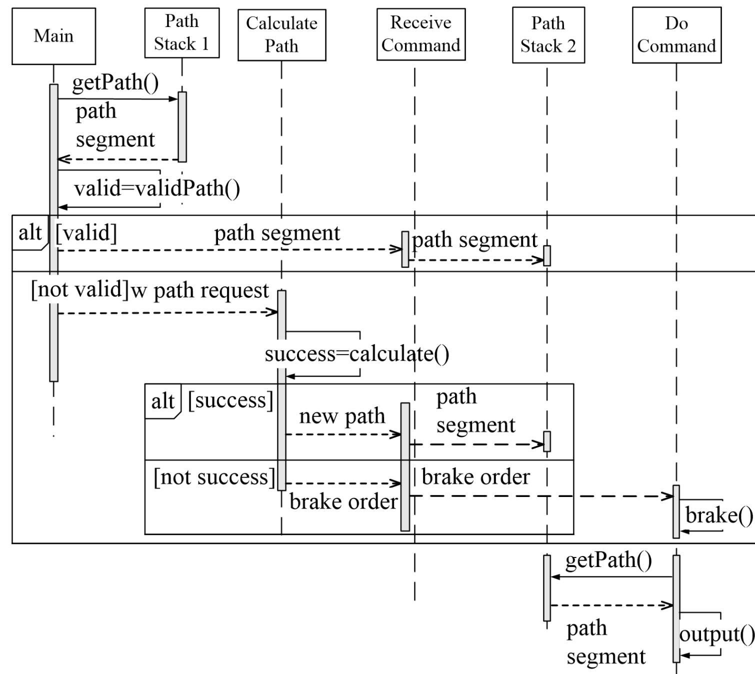


Figure 8.4: Sequence diagram of tasks in the control system

Calculate Path Task will send a braking command to *Receive Command Task*, which then stores the path segment into *Path Stack 2* where *Do Command Task* gets path segments. In the end, *Do Command Task* generates an output to the actuator, based on the commands.

System Requirements. The AWL has a large set of functional and extra-functional requirements. Below, we present some of these requirements, which are formally verified in this paper.

1) **Initial path computation**: during initialization, the AWL must compute an initial path to the destination, which must avoid all the static obstacles identified in the quarry;

2) **Obstacle avoidance and path recalculation**: the AWL must avoid static and dynamic objects around it in due time before returning to the initial path;

3) **Mode switch**: when a critical error occurs (e.g., an obstacle cannot be safely avoided or be reported to the control unit), the AWL must switch to the safety mode in order to freeze all motions within a certain time limit, to avoid further damage. In this case, the reaction time limits are error-specific;

4) **End-to-end deadline**: To guarantee a certain productivity, the AWL must reach the destination within 2200 milliseconds.

8.3 Preliminaries

In this section, we overview the background information needed for the rest of the paper: timed automata and UPPAAL, as well as the A* and dipole flow field algorithms.

8.3.1 Timed Automata and UPPAAL

UPPAAL [9, 1] is a tool suite for modeling, simulation, and model checking of real-time systems. The modeling formalism of UPPAAL is an extension of *timed automata* (TA) [10], which is defined as the following tuple:

$$< L, l_0, A, V, C, E, I > \tag{8.1}$$

where: L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $A = \Sigma \cup \tau$ is a set of *actions*, where Σ is a finite set of *synchronizing actions* and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization, V is a set of *data variables*, C is a set of *clocks*, $E \subseteq L \times B(C, V) \times A \times 2^C \times L$ is the set of *edges*, where $B(C, V)$ is the set of *guards* over C and V, that is, conjunctive formulas of clock constraints ($B(C)$), of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and non-clock constraints over V ($B(V)$), and $I : L \longrightarrow B_{dc}(C)$ is a function that assigns *invariants* to locations, where $B_{dc}(C) \subseteq B(C)$ is the set of downward-closed clock constraints with $\bowtie \in \{<, \leq, =\}$. The invariants bound the time that can be spent in locations, hence ensuring progress of TA's execution. An edge from location l to location l' is denoted by $l \xrightarrow{g,a,r} l'$, where g is the guard of the edge, a is an update action, and r is the clock reset set, that is, the clocks that are set to 0 over the edge.

In UPPAAL, locations are marked as *urgent* (denoted by encircled u) or *committed* (denoted by encircled c), indicating that time cannot progress in such locations. Committed locations are more restrictive, requiring that the next edge to be traversed needs to start from a committed location. Variables

and clocks can be set to certain values by the updates along the edges. In UPPAAL, an update can be a comma-separated list of expressions, or a C-code style function that is implemented in the declaration of TA.

The semantics of TA is a *labeled transition system*. The states of the labeled transition system are pairs (l, u), where $l \in L$ is the current location, and $u \in R^C_{\geq 0}$ is the clock valuation in location l. The initial state is denoted by (l_0, u_0), where $\forall x \in C,\ u_0(x) = 0$. Let $u \models g$ denote that clock value u satisfies guard g. We use $u + d$ to denote the time elapse where all the clock values have increased by d, for $d \in \mathbb{R}_{\geq 0}$. There are two kinds of transitions $\rightarrow$:

(i) Delay transitions: $< l, u > \xrightarrow{d} < l, u + d >$ if $u \models I(l)$ and $(u + d') \models I(l)$, for $0 \leq d' \leq d$, and

(ii) Action transitions: $< l, u > \xrightarrow{a} < l', u' >$ if $l \xrightarrow{g,a,r} l', a \in \Sigma, u \models g$, clock valuation u' in the target state (l', u') is derived from u by resetting all clocks in the reset set r of the edge, such that $u' \models I(l')$.

TA are composed into a *network of TA* over a common set of clocks and actions [2]. In this paper, we model the communication between TA via synchronization channels (e.g., a! and a?) with rendezvous semantics: a sender (a!) synchronizes with a receiver (a?), provided that the sending and receiving edges are enabled, that is, their guards are satisfied. The UPPAAL model checker supports the verification of queries written in a decidable subset of Timed Computation Tree Logic (TCTL) [2]. The syntax of a TCTL formula consists of quantifiers over paths and path-specific temporal operators. There are two types of path quantifiers: the universal one, "A" meaning "for all paths", and the existential one, "E" denoting "there exists a path". We are interested in two path-specific temporal operators, that is, "$Always$" ($\Box$) temporal operator meaning that a given formula is true in all states of a path, and the "$Eventually$" ($\Diamond$) operator meaning that a formula becomes true in finite time, in some state along a path. The UPPAAL queries that we verify in this paper are properties of the form: (i) **Invariance**: $A\Box p$ means that for all paths, for all states in each path, p is satisfied, (ii) **Liveness**: $A\Diamond p$ means that for all paths, p is satisfied by at least one state in each path, (iii) **Reachability**: $E\Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (iv) **Time-bounded Leads to**: $p \leadsto_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter; it is equivalent to the property: $A\,\Box\,(p \Rightarrow A\,\Diamond_{\leq t}\,q)$.

Algorithm 1: A* Algorithm

Input: Node $start$, Node $destination$
Output: If the path is found or not

1. $closed := open := \varnothing$
2. $parent(start) := start$
3. $g(start) := 0$
4. $open.Insert(start, g(start) + h(start))$
5. **while** $open \neq \varnothing$ **do**
6. $current := open.top()$ /*return and remove the node with the minimum cost in open*/
7.   **if** $current = destination$ **then**
8.   return "arrived"
9.   **end**
10.   closed.Insert(current)
11.   **foreach** $n \in neighbors(current)$ **do**
12.   **if** $n \notin closed$ **then**
13.   **if** $n \notin open$ **then**
14. $g(n) := \infty$
15. $parent(n) := NULL$
16.   **end**
17. $g_{old} := g(n)$
18.   **if** $g(current) + c(current, n) < g(n)$ **then**
19. $parent(n) = current$
20. $g(n) = g(current) + c(current, n)$
21.   **end**
22.   **if** $g(n) < g_{old}$ **then**
23.   **if** $n \in open$ **then**
24. $open.Remove(n)$
25.   **end**
26. $open.Insert(n, g(n) + h(n))$
27.   **end**
28.   **end**
29.   **end**
30.   return "no path found"
31. **end**

8.3.2 A* Algorithm

The A* algorithm is a widely used algorithm for path finding and graph traversal [7], and it was first introduced by Hart et al. [11]. In this paper, we use it to compute the initial path for AWL. It is an extension of Dijkstra's algorithm that uses a heuristic function to guide the graph traversal in order to achieve better performance. The basic idea of the A* algorithm is to find a lowest cost path from all possible paths to the destination, similar to Dijkstra's algorithm. While exploring the graph, the cost of the current node is calculated by the following function: $f(n) = g(n) + h(n)$, where n is the current node, $g(n)$ is the cost from the starting node to n, and $h(n)$ is the estimated cheapest cost from n to the destination. Intuitively, the A* algorithm aims to find the path that minimizes $f(n)$.

The pseudo code of the A* algorithm [12] is shown by Algorithm 1. It works in weighted graphs and constructs a tree of paths starting from a specific node of the graph, which is defined as the input. From line 1 to line 4, two arrays are initialized, that is, $open$: the set of currently discovered nodes that are not evaluated yet, and $closed$: the set of nodes that have been evaluated already. From lines 5 to 16, the main loop starts, in which the node with the minimum cost in $open$ is selected. If this node is the destination, the calculation ends. Otherwise, the neighbors of this node and denoted by n, which are one-cell distance away around the node, are considered one by one as candidates to the $open$ set, and evaluated in the rest of the code. From lines 17 to 21, the cost of node n is updated to the minimum and its parent node is changed accordingly. And between lines 22 to 26, the open set either updates the cost of node n, or inserts a new node n and its cost into the set.

8.3.3 Dipole Flow Field for Collision Avoidance

Modeling the paths of moving vehicles or other dynamic objects is not an easy task. Some studies have adopted the so-called *static flow field* and *dynamic dipole field* algorithms to represent the interactions of such moving objects [8]. In such scenarios, a vehicle moves within a certain area, called the map, and travels along a preset path that avoids the static obstacles, and approaches the destination. As soon as it discovers a moving obstacle within its vision range, the vehicle runs the collision avoidance algorithm to stay away from the obstacle as well as move towards the destination.

In this case, the static flow field force attracts the vehicle to its goal, ensuring that the vehicle avoids the static obstacles on the map. Meanwhile, as soon

as a dynamic obstacle is encountered (be it another moving vehicle or a human), the dynamic dipole field algorithm generates forces that push the vehicle away from the dynamic obstacle, based on the latter's respective moving direction and velocity when within a close range from the original vehicle. The static flow field force is calculated by the following equation: $F_a = \frac{k_a q_0 Q}{D^2}$, $F_r = \frac{k_r q_0 q_1}{d^2}$, and $F_{\text{flow}} = F_a + F_r$, where F_a is the attractive force that draws the vehicle back to its initial path, F_r is the repulsive force from the nearby static obstacles, k_a, k_r, q_0, q_1, Q are coefficients whose values are problem specific, whereas D and d are the distances between the vehicle and its goal, and between the vehicle and the static obstacle, respectively. Unlike the dipole field forces, the attraction and repulsive forces always exist, regardless of whether the vehicle is moving or not.

In the theory of dipole field, every object is assumed to be a source of magnetic dipole field, in which the magnetic moment is aligned with the moving direction, and the magnitude of the magnetic moment is proportional to the velocity. Concretely, the repulsive force of a moving obstacle acting on the vehicle can be formulated as follows:

$$\vec{m} = k_m \vec{v} \tag{8.2}$$

$$\begin{aligned} \vec{F_d} = \frac{k_d}{d^5}[(\vec{m_0} \cdot \vec{r}) \times \vec{m_i} + (\vec{m_i} \cdot \vec{r}) \times \vec{m_0} + (\vec{m_0} \cdot \vec{m_i}) \times \vec{r}] - \\ \frac{5 \cdot (\vec{m_0} \cdot \vec{r}) \cdot (\vec{m_i} \cdot \vec{r})}{d^2} \times \vec{r}], \end{aligned} \tag{8.3}$$

where $\vec{r}$ is the distance vector between the two objects ($k_m, k_d \in R^+$). The combination of the static flow field and the dynamic dipole field ($F = F_{\text{flow}} + F_d$) guarantees that the vehicle moves safely by avoiding all detected obstacles, and reaches the destination eventually as long as the path is safe.

8.4 AWL's Modeling and Verification

In this section, we present the formal model of the AWL, as a network of TA, and the verification results after employing UPPAAL on the formal model. The model consists of three parts: the map, the AWL's movements, and the AWL's control system. Figure 8.5 depicts the verification methodology proposed in this paper. First, the map is modeled as a data structure. Next, the movements of dynamic obstacles and AWL, which include straight moving, turning and braking, are designed, assuming the actors are functionally correct in the given map. Then, we model the AWL's control system as a network of timed automata, in which tasks are TA that communicate via shared global variables

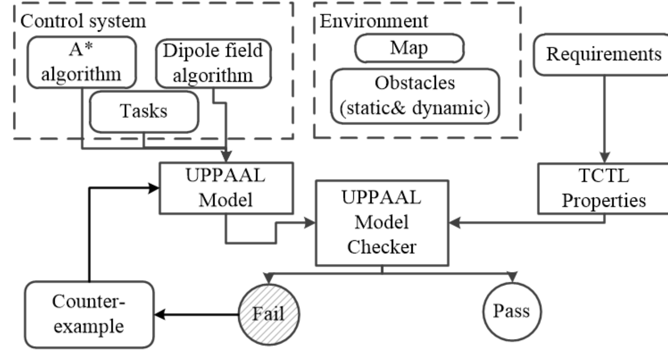


Figure 8.5: AWL's modeling and verification process

and synchronize via channels. The UPPAAL model checker is then applied to verify whether the timed automata network satisfies the AWL requirements that are formalized as TCTL properties.

8.4.1 Map Abstraction

The loader's working environment consists of a map and several obstacles. The map is abstracted into a 2-dimensional Cartesian grid of disjoint cells with resolution $\epsilon \in \mathbb{R}_{+}$. As Figure 8.6 shows, the location of an object on the map is denoted by (x, y), with $x, y \in \mathbb{R}_{\geq 0}$.

The grid is encoded as (z_x, z_y), with $z_x, z_y \in \mathbb{Z}_{\geq 0}$. The mapping from reals to integers on two axes is given by the following:

$$
\begin{gathered}
f_1 : \mathbb{R}_{+}^2 \rightarrow \mathbb{Z}_{+}^2 \quad f(x, y) = (z_x, z_y) \\
\text{if } x - \frac{\epsilon}{2} \leqslant z_x \leqslant x + \frac{\epsilon}{2}, \text{ and } y - \frac{\epsilon}{2} \leqslant z_y \leqslant y + \frac{\epsilon}{2}
\end{gathered} \tag{8.4}
$$

If an object (static or dynamic) is located at the intersection of x and y axes, the object's position is marked by $\times$ as shown in Figure 8.6. If the intersection is occupied, no other object can move to that point anymore. In our model, each intersection point is assigned 0 or 1, denoting that the point is empty or occupied, respectively. Furthermore, we assume that a dynamic obstacle occupies one point only, whereas a static obstacle can occupy more than one point as one can see in Figure 8.6. Based on this abstraction, the map is defined as a 2-dimensional array in UPPAAL, where each element represents a point on the map, and is assigned 0 or 1. A vertex is defined as a structure *Vertex* with two elements, integers x and y, representing the coordinates on x and y

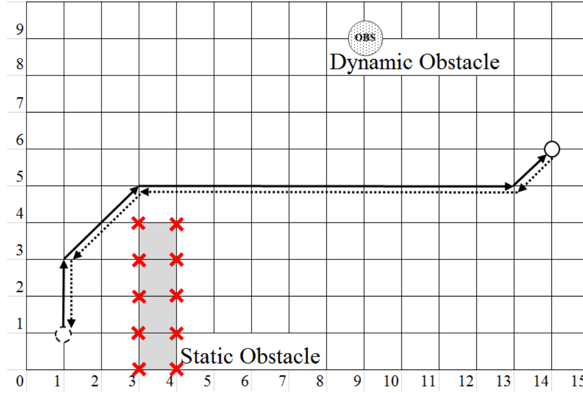


Figure 8.6: Abstraction of the map

axes, respectively; the static obstacle is defined as a constant array of *Vertex*, representing the coordinates of each of its vertices (see Code 8.1).

Code 8.1: Vertex and static obstacle definitions

```
const int N = 15;
typedef struct
{
   int[0, N] x;
   int[0, N] y;
}Vertex;
const Vertex staticObstacle[10] =
     {{3,0},{3,1},{3,2},{3,3},{3,4},{4,4},{4,3},{4,2},{4,1},{4,0}};
```

8.4.2 Movements Abstraction

As the objects' locations are mapped onto the line intersections in the map, their movements are then restricted to the edges or the diagonals of the cells, as depicted in Figure 8.6. In our model, we separate the path into several path segments that are defined as pairs of vertices. A vertex is denoted by (z_x, z_y) as in formula (8.4), whereas v denotes the velocity of the AWL. Consequently, the path is defined as a sequence of path segments:

$$p = (z_{x_0}, z_{y_0})(z_{x_1}, z_{y_1}) \cdots (z_{x_{n-1}}, z_{y_{n-1}})(z_{x_n}, z_{y_n}) \tag{8.5}$$

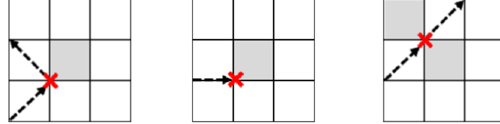


Figure 8.7: Three types of forbidden movements

$$\begin{cases} z_{x_i} = z_{x_{i-1}} \pm v, \text{ where } x_i \geq 1 \\ z_{y_i} = z_{y_{i-1}} \pm v, \text{ where } y_i \geq 1 \end{cases} \tag{8.6}$$

As mentioned previously, the AWL cannot occupy the vertices of a static obstacle, as shown in Figure 8.7.

When the loader starts to move, it accelerates from the minimum velocity (modeled as 0) to the maximum velocity (modeled as 2). The AWL stays at the current position for 2 time units at speed 0, then the duration decreases by 1 as the speed increases by 1, until it reaches the maximum velocity. The time unit is the execution period of *Main Task* in the control unit of the AWL.

We model the dynamic obstacle as a TA in UPPAAL, with a self-looping location that encodes the movements of the obstacle. The changing position of the obstacle is implemented by a function executed when the self-loop edge is traversed.

8.4.3 Formal Model of AWL's Control System

As shown in Figure 8.2, the control system consists of three units: vision unit, execution unit, and control unit. The vision unit acquires data from LIDAR and executes the recognition algorithms to identify the shapes, types, moving directions, etc., of the obstacles. However, in our model, we do not include

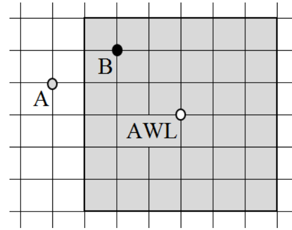
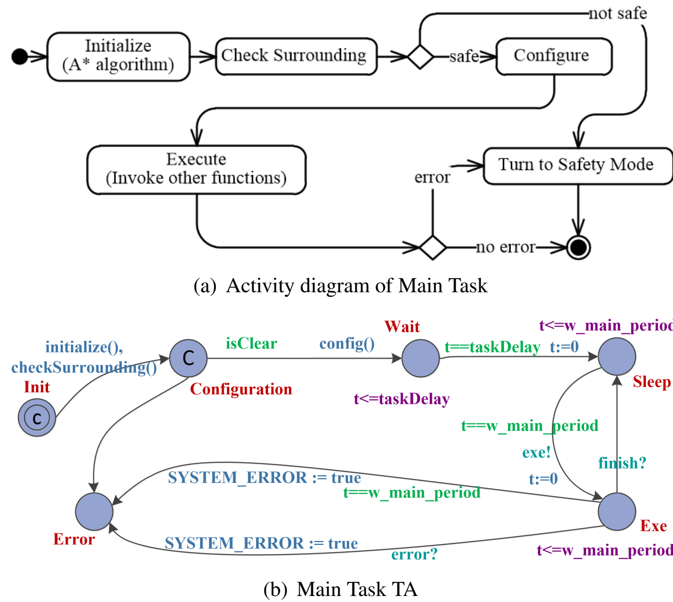


Figure 8.8: Detection range of the AWL



(a) Activity diagram of Main Task

(b) Main Task TA

Figure 8.9: Modeling Main Task from the activity diagram

this algorithm per se. Instead, our model checks the values of the map's points (0 or 1) to detect the obstacles present in the vicinity of the AWL, assumed as an area of maximum 3-cells distance from the AWL. In Figure 8.8, we see that object A cannot be detected as it is out of range, but object B is reported as an obstacle.

To model the AWL's control system, we create a TA for each task and function presented in Figure 8.3, whose procedures are captured by activity diagrams (e.g., Figure 8.9(a)) and sequence diagrams (e.g., Figure 8.4). We map the elements of such diagrams (e.g., decision nodes and action nodes in the activity diagrams) to our model, so that each TA's structure is respectively constructed. Even if the TA are manually created, we have used a 1-to-1 mapping in this process, described as follows:

1. For each action node of the activity diagram (except for those that call other functions), we create functions and corresponding locations and edges to ensure the same order of execution of the respective task, as in the original diagram.

2. Decision nodes of the activity diagram are represented as locations with

multiple outgoing edges, with edges enabled based on the associated guards.

3. For action nodes that call other functions, we use synchronization among TA to model the invoking relation. This step is elaborated in the following examples.

4. After the structure of the TA is constructed, we implement the A* and dipole field algorithms as C-code functions in the TA, respectively.

The final model contains 12 TA (i.e., 11 TA for the tasks and functions, 1 for the dynamic obstacle), and 61 C-code functions. Due to space limitation, we select to describe the respective behaviors and communication among *Main Task*, *Calculate Path Task*, and *Get Path Function* TA.

According to *Main Task*'s activity diagram of Figure 8.9(a), the first two action nodes aim to initialize the system, check its surroundings, and run the A* algorithm. Hence, two locations, namely *Init* and *Configuration*, and the edge connecting them are created in the *Main Task* automaton shown in Figure 8.9(b). Along the edge, two functions, namely *initialize()* and *checkSurrounding()*, initialize the system's variables and check for obstacles around AWL, when executed. The A* algorithm is also executed in *initialize()* to generate the initial path. Next, two mutually exclusive outgoing edges from location *Configuration* are added, corresponding to the decision node in Figure 8.9(a) that follows the action node *Check Surrounding*, and indicating that if some error occurs during the initialization, the system moves to location *Error* to freeze the AWL. In case of no error, it moves to location *Wait*, where the task waits to be invoked. The other TA (e.g., Figure 8.10 and 8.11) are constructed by following similar steps.

After representing each individual task by a corresponding TA, the communication and scheduling of tasks need to be modeled. Every task has an execution period and is scheduled in a certain order. To achieve this, we add extra locations and invariants in the model, which are not corresponding to the action nodes and decision nodes of the activity diagram, such that some TA are executed periodically. The tasks for detecting obstacles and acquiring positions must be started earlier and executed more frequently than other tasks, such that the control system always makes decisions on the latest information. Hence, as it is shown in Figure 8.9(b), the automaton of *Main Task*, which awaits position information from other tasks, stays at location *Wait* until clock t reaches the value of *taskDelay*, which is 7 in this case. This delay enables *Main Task* to start later than the tasks using the system clock. To model the period of the

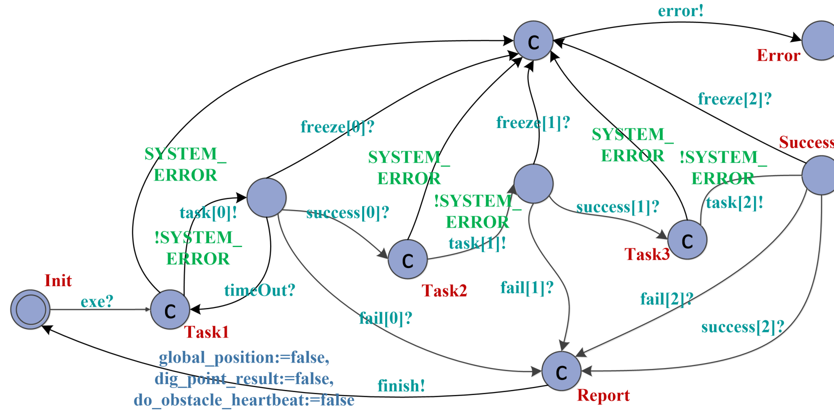


Figure 8.10: Timed automaton of Execution function

task, after it moves to location *Sleep*, the automaton waits again until clock *t* reaches its task period, i.e., constant integer *w_main_period*, when it can be executed.

According to the sequence diagram of Figure 8.4, *Main Task* calls sub-functions. Hence, the automaton of *Main Task* is synchronized with *Execution Function*, via channel *exe* that decorates the edge connecting locations *Wait* and *Exe*. Then *Main Task* delays in location *Exe* until it synchronizes again with *Execution Function* via channel *finish* or *error*, indicating that the work is done or some error occurs.

The automaton *Execution Function* is also synchronized with other automata for the same reason. For instance, as shown in Figures 8.10 and 8.11, on channel *task[0]*, the automaton *Get Path Function* is synchronized with *Execution Function*, indicating that *Get Path Function* is called by *Execution Function*. In addition, waiting for data from another task is modeled by locations and invariants added to *Get Path Function* automaton. For example, Figure 8.11 depicts that *Get Path Function* automaton waits for position data (*global_position*), in location *Wait* until clock *w_task1_trigger* reaches its limit *w_task1_ threshold*, when both *w_task1_trigger* is set to *w_task1_threshold* and variable *global_position* is set to *true* by other TA, indicating that the AWL's position has been acquired, or the variable *global _position* remains *false* until the invariant is violated; if the latter, the automaton moves back to the initial location *Start*, meaning that a time-out event occurs in *Get Path Function*.

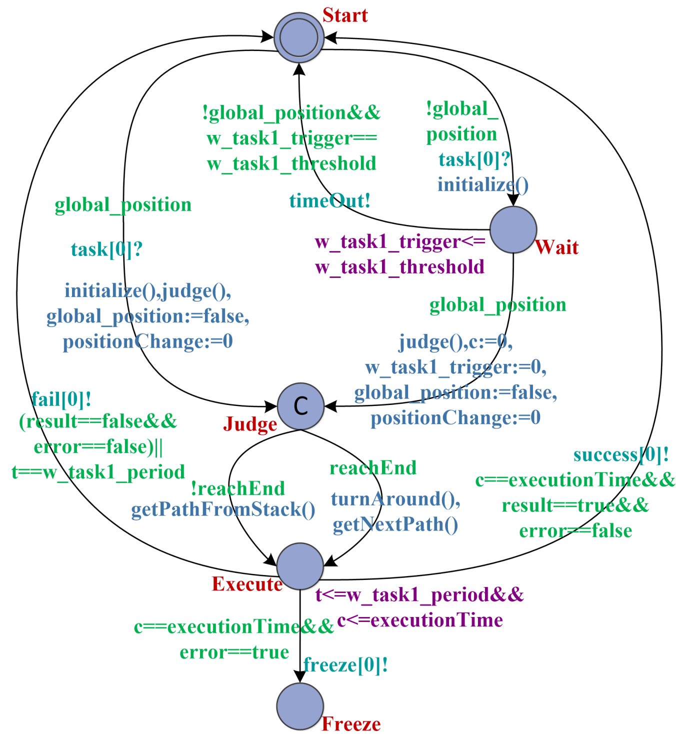


Figure 8.11: Timed automaton of Get-Path function

The dipole field algorithm is implemented as C functions in the automaton *Calculate Path Task*, as shown in Figure 8.12. The task is executed in case an obstacle is detected, or AWL deviates from the initial path, so in the task's automaton, the first function being executed after *initialize()* is *findNextPosition()*, where the grid point in the initial path closest to the current position is returned as the new next position, which cannot be ensured to be safe. Hence, the forces applied on the AWL are computed in function *calculateForces()* based on the new position and equations described in Section 8.3.3. After that, a new path segment, if it exists, is calculated in function *calculateNewPath()* according to the field forces, which guarantees the safety of the AWL. The implementation is explained in detail in Section 8.5. If the new path segment does not exist (*newPathCorrect*==*false*), the automaton moves to location *Error* and sends out the *brake command* (*brake_request_udp* := true).

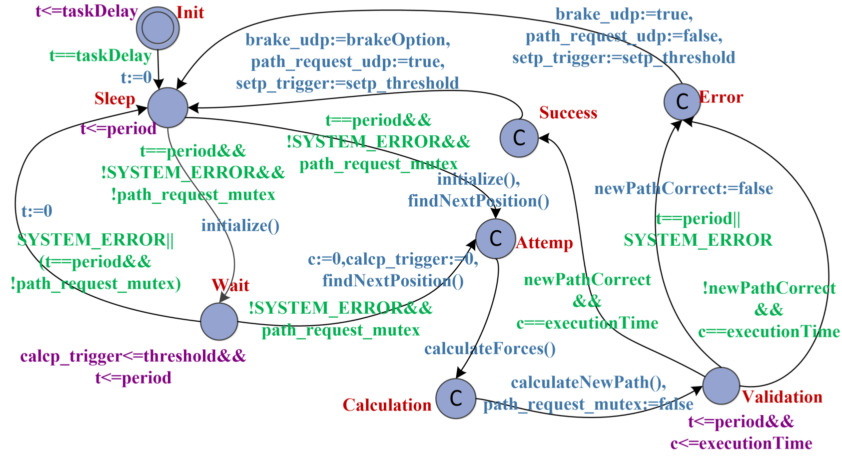


Figure 8.12: Timed automaton of Calculate Path task

8.4.4 AWL's Model Verification

By applying the modeling process described in Sections 8.4.1, 8.4.2, and 8.4.3, we create the formal model of the AWL and its environment as a network of TA. As mentioned, the formal model consists of 12 TA (11 TA for the tasks and functions presented in Figure 8.3, plus one TA for the dynamic obstacle), four data structures (one for the map, two for the A* algorithm, and one for the path stack, which is used to store the path segments), 23 clocks, 49 global variables, 61 C-code functions, etc. To verify whether this model satisfies the informal requirements given in Section 8.2, we formalize the latter as TCTL queries that we check with UPPAAL. Two versions of the map are used in the verification. As depicted in Figure 8.13, we use a map with a static obstacle that occupies 10 grid points, and where the stone pile and crusher are located at (1,1) and (14,6), respectively. Next, as shown in Figure 8.14, we add one dynamic obstacle to this map, which starts at point (9,8) and moves along a predefined path.

Table 8.1 presents the TCTL queries and the verification results. In the rest of this section, we describe these results.

Initial path computation. To verify that the AWL cruises between the stone pile and the crusher, and avoids the static obstacle, we use the map of Figure 8.13. Seven queries are specified to verify this requirement.

Queries Q1.0 and Q1.1 require that the initial path is calculated after the

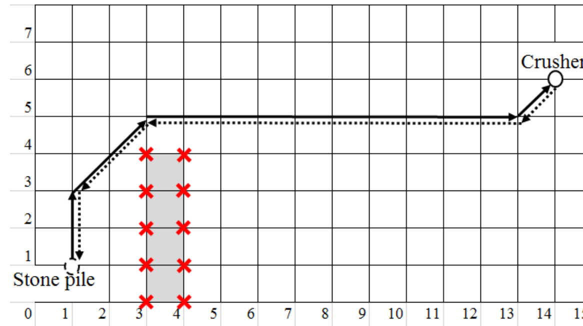


Figure 8.13: The AWL's trajectory on the map with a static obstacle

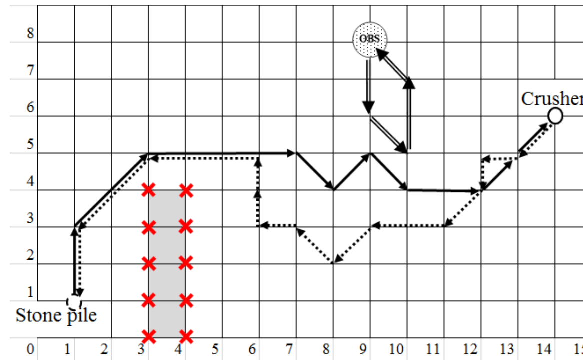


Figure 8.14: The AWL's trajectory on the map with a static and a dynamic obstacle

automaton *mainTask* moves to location *Wait* (mainTask.Wait). The integer variable *lenOfPathStack*, whose initial value is 0, is assigned by the length of the path stack, where the initial path is stored. Once the variable becomes greater than 0, the initial path is generated. Query Q1.3 states that, if the AWL is at the stone pile (currentPosition == pile) and its destination is the crusher (destination == crusher), the AWL will indeed eventually reach the crusher (currentPosition == crusher). Since UPPAAL's "leads to" operator ($p \leadsto q$) is equivalent to $A\Box(p \Rightarrow A\Diamond q)$, we first check in query Q1.2 if the antecedent of Q1.3, that is, (currentPosition == pile and destination == crusher) is reachable. In our scenario, the AWL's initial location is the stone pile and the target

Table 8.1: Verification queries and results

Requirement	Query	Result	States explored	Time (ms)
Initial path computation	Q1.0: $E <>$ mainTask.Wait	Pass	2	110
	Q1.1: $A <>$ mainTask.Wait imply lenOfPathStack > 0	Pass	8780	484
	Q1.2: $E <>$ currentPosition == pile and destination == crusher	Pass	1	0
	Q1.3: (currentPosition == pile and destination == crusher) $-->$ currentPosition == crusher	Pass	14191	1125
	Q1.4: $E <>$ currentPosition == crusher and destination == pile	Pass	2339	297
	Q1.5: (currentPosition == crusher and destination == pile) $-->$ currentPosition == pile	Pass	14204	782
	Q1.6: $A[]$ forall(i:int[0,9]) currentPosition != staticObstacle[i]	Pass	8780	485
Obstacle avoidance	Q2.0: $A[]$ currentPosition != currentObstacle	Pass	125941	6297
	Q1.3: (currentPosition == pile and destination == crusher) $-->$ currentPosition == crusher	Pass	227646	13969
	Q1.4: $E <>$ currentPosition == crusher and destination == pile	Pass	2678	375
	Q1.5: (currentPosition == crusher and destination == pile) $-->$ currentPosition == pile	Pass	192406	10656
Mode switch: error A	Q3.1: $E <>$ errorStart == true	Pass	30	234
	Q3.2: error_start==true $-->$ (SYSTEM_ERROR==true and reaction_time$<=$20)	Pass	91	250
Mode switch: error B	Q3.1: $E <>$ errorStart == true	Pass	29	234
	Q3.2: error_start==true $-->$ (SYSTEM_ERROR==true and reaction_time$<=$15)	Pass	320	266
End-to-end deadline	Q4.0: (currentPosition==pile and destination==crusher) $-->$ (currentPosition==pile and destination==pile and gClock $<=$ 2200)	Pass	590326	36641

is the crusher, thus query Q1.2 is satisfied by the initial state of the model and the verification explores only one state, the initial state. Similarly, in queries Q1.4 and Q1.5 we verify whether the AWL moves back from the crusher to the stone pile, whereas in query Q1.6 we check that the autonomous loader

avoids the static obstacle. Concretely, query Q1.6 requires that the AWL must never occupy one of the grid points of the static obstacle (A $\square$ forall(i:int[0,9]) currentPosition != staticObstacle[i]).

The combination of the first seven queries of Table 8.1 verifies that the autonomous loader cruises safely between the stone pile and crusher and back, without colliding with the static obstacle, thus showing that the AWL has the ability to compute a safe initial path. Furthermore, in order to visualize this path, we use the following queries:

E $\lozenge$ currentPosition == pile and destination == pile
E $\lozenge$ currentPosition == crusher and destination == crusher

These queries require that there exists at least one execution path in which the AWL eventually reaches the stone pile and the crusher, respectively. They are weaker than queries Q1.3 and Q1.5, but for these queries, the model checker generates a witness trace, which represents the initial path. The path presented in Figure 8.13 is generated in this way.

Obstacle avoidance and path recalculation. To verify this requirement, one dynamic obstacle (e.g., another vehicle) is added to the map, as shown in Figure 8.14. We assume that this object is not equipped with an obstacle avoidance feature, thus it does not change its path when it approaches the AWL. To verify this requirement, we need to check again that the AWL can reach the crusher and the stone pile, respectively, that is, queries Q1.2 to Q1.4, assuming the updated map. In query Q2.0, we check that AWL does not collide with the dynamic obstacle (A $\square$ currentPosition != currentObstacle) for all possible execution paths of the model. As presented in Table 8.1, the number of states explored and the verification time for queries Q1.3, Q1.4 and Q1.5, in this case, are drastically increased as compared to the initial path computation case, since the dynamic obstacle increases the complexity of the model. As previously, we generate the path followed by AWL, which is depicted in Figure 8.14. The solid arrows represent the path to the crusher, the dashed arrows represent the way back to the stone pile, and the double-line arrows represent the preset path of the dynamic obstacle.

Mode switch. This requirement is verified on the map that contains the dynamic obstacle. To verify that AWL switches to safety mode and freezes its motion whenever it malfunctions, we introduce a global boolean variable `SYSTEM_ERROR` that "injects" errors into the model. For instance, in Figure 8.10, the TA moves to location `Error` whenever `SYSTEM_ERROR` becomes true. To check that this variable never evaluates to true, we use the following query:

$$A \square \, !SYSTEM_ERROR$$

As expected, this query is satisfied unless we inject faults into the model. In this paper, we model two faults that mimic real malfunctions:

- Error A: we set the boolean variable *do_obstacle_heartbeat* in the *Do Obstacle Task* to false, when it is sending a message and is resetting the clock *reaction_time* to zero at the same time, indicating that the information on obstacles cannot be reported to the control unit, which can be very dangerous.

- Error B: we set the boolean variable *position_udp* in the *Position Task* to false when it is sending the position information through the Ethernet, implying that the information is lost during the transmission.

For both these errors, two queries (Q3.1 and Q3.2) are formulated for verification. Query Q3.1 checks that there exists at least one execution path in which the error eventually happens (E $\Diamond$ errorStart == true). Formula Q3.2 requires that once an error occurs (`SYSTEM_ERROR` == true), the system must detect and react to the error within a certain time bound. This time bound is 20 time units for error A (reaction_time $<=$ 20 in Q3.2 A), and 15 times units for error B (reaction_time $<=$ 15 in Q3.2 B). The verification results show that, when error A or error B occur, the system moves to the `SYSTEM_ERROR` mode within the required time bound.

End-to-end deadline. The autonomous wheel loader must not only be able to travel to the crusher and then return to the stone pile position, but it also needs to accomplish this task within a certain time bound, which is its end-to-end deadline of 2200 time units. This requirement is verified by query Q4.0, which is a time-bounded leads to property, whose antecedent (that is, currentPosition == pile and destination == crusher) is the initial state of the model, and the consequent requires AWL to return to the stone pile before the deadline (currentPosition == pile and destination == pile and gClock $<=$ 2200). Since Q4.0 is a leads-to property, we also need to verify that its antecedent is reachable, by proving Q1.2. Furthermore, by checking the query:

$$E\Diamond \text{ currentPosition==pile and destination==pile},$$

we can request the model checker to generate the fastest diagnostic trace, which gives us the fastest time (1620 ms) to complete one cruise.

8.5 Discussion

The issue with our abstraction of movements is that the shortest path that A* algorithm generates in the discrete area is not equivalent to the shortest path in the continuous area, because it constrains paths to be formed by the edges or diagonals of the cells. Some other path-planning algorithms, e.g., Theta* algorithm, overcome this drawback by changing the path to an any-angle path that does not necessarily follow the edges of the cells[12]. However, as traditional UPPAAL only supports integers, it is very difficult to implement algorithms like Theta* or dipole field as such, therefore they must be simplified. Hence, in our model, the forces in the dipole field algorithm that is employed by AWL for collision avoidance, are calculated using integers, based on the formulas in Section 8.3.3. Moreover, instead of using Newton's law of motion, which involves real numbers to calculate the loader's position, we use the sign of the combination of forces to decide the next position. Formula 8.7 shows the relation between the signs of forces and the AWL position, where (x',y') models the next position of AWL, (x,y) represents the current coordinates of AWL, F_x, F_y model the combination of attractive and repulsive forces on x axis and y axis, respectively, and T is the threshold for movements. This formula restricts the AWL to move only along the edges or diagonals of the cells, which is exactly our abstraction for movements.

$$(x', y') = \begin{cases} (x+1, y+1), \text{ if } F_x \geq T, F_y \geq T \\ (x+1, y), \text{ if } F_x \geq T, F_y < T \\ (x, y+1), \text{ if } F_x < T, F_y \geq T \\ (x, y), \text{ if } F_x < T, F_y < T \end{cases} \quad T \in \mathbb{Z}^+ \tag{8.7}$$

To fully implement the dipole field algorithm, a tool that fully supports floating point numbers is desirable. UPPAAL SMC (Statistical Model Checker) satisfies this requirement while still enjoying most of the useful features of UPPAAL [13]. With UPPAAL SMC, we can also model stochastic behaviors, e.g., the occurrence of dynamic obstacles, the reliability problem of Ethernet, etc. However, UPPAAL SMC does not provide exhaustive model checking, for it provides the probability of satisfying the queries.

We have also verified the AWL model in different scenarios, e.g., by letting the dynamic obstacle move arbitrarily within the map rather than along a preset path. It turns out to be very difficult to satisfy the requirements of reaching the destination while avoiding the obstacle under such circumstances. Two scenarios are generated by UPPAAL, where the AWL either collides with the

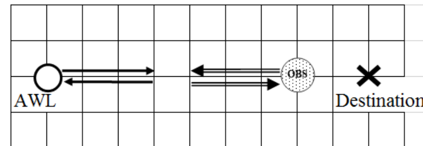


Figure 8.15: A livelock scenario

dynamic obstacle or is stuck into a "livelock". As depicted in Figure 8.15, the AWL and the obstacle move back and forth on the same axis because there is no force on the other axis that turns the AWL at an angle to the obstacle. Therefore, the AWL consistently moves back and forth on this axis but never gets to the destination.

In another scenario the obstacle keeps “pushing” the AWL until both of them reach the edge of the map and stop, on grounds of our assumption that the dynamic obstacle does not avoid the AWL even though they come close to each other. One possible solution is to optimize the implementation of the dipole field algorithm so that the AWL can actively and angularly move towards the obstacle’s moving direction, such that the AWL will bypass the obstacle from behind instead of being pushed away by the obstacle.

8.6 Related work

A number of formal methods have been applied to the verification of autonomous vehicles. Saberi et al. [6] propose using high-level languages, namely mCRL2 and Modal μ-calculus, for specifying and verifying multi-robot systems. Smith et al. [14] propose a method, based on weighted transition systems and Buchi automata, to find the optimal trajectory for the robot, which satisfies the requirements described in LTL. Koo et al. [15] propose a framework for the coordination of a network of autonomous robots with respect to formal requirements specifications in temporal logics, in which hybrid automata and Cadence’s SMV model checker are used. Quottrup et al. [16] [3] design a high level abstraction of a multi-robot system using timed automata. Their model consider 4-directions movements of autonomous robots. Moreover, they also generate the shortest path with UPPAAL. Our research is inspired by such studies and provides modeling and verification of a complex control system of an autonomous wheel loader. However, instead of using a model checker to generate paths, we employ intelligent algorithms for path planning and collision avoidance (via dipole field) that we formally verify together with the control

system.

There are also different approaches for modeling and analyzing the path-planning algorithms of autonomous vehicles. Fainekos et al. [17] apply temporal logic and model checking tools to generate discrete path plans that are later translated to continuous trajectories using hybrid control. The approach that they propose is built upon an existing framework [18] and proves that discrete plans and continuous trajectories are bisimilar, so that the satisfaction of LTL properties on the former is preserved by the latter. Kripke models and model checking techniques have been employed by Jeyaraman et al. in their study of modeling and verification of cooperative unmanned aerial vehicle (UAV) teams [19]. Rabiah et al. [20] use the Z specification language to formally specify the A* path planning algorithm, and verify the correctness of the algorithm by theorem proving. Saddem et al. [21] also use UPPAAL and CTL to verify reachability properties of autonomous behavior, including path finding. They propose a decomposition methodology to reduce the memory requirement and execution time of model checking. What makes our work different from the above studies is that we carry out a more extensive verification of the autonomous vehicle against a rich set of complex and realistic safety properties expressed in TCTL, e.g., whether the system can react to an error within a certain time limit. In addition, our formal model is more detailed, including the tasks in the control system and their communication, the algorithms, acceleration and deceleration of the vehicle. The conjunction of all these elements increases the size of model's state space dramatically, and hence the complexity of verification.

Some studies focus on the formal modeling and verification of the control logic or internal architecture of the automation system. Chouali et al. [22] propose an approach to model and ensure formally the reliability of automotive applications. They use SYSML to model the system before verifying the model described using interface automata. Hanisch et al. [23] [24] adopt the Net Condition/Event Systems (a modular extension of Petri nets) in their modeling and verification of several automated systems in intelligent manufacturing area. In comparison to these studies, our model includes not only the components in the control system but also the behaviors of the AWL and its environment, which allows us to simulate the model in a reactive mode, and our verification includes properties that are crucial for real-time automotive systems (e.g., end-to-end deadlines).

8.7 Conclusions

In this paper, we have presented the formal modeling and verification of an industrial prototype of an Autonomous Wheel Loader equipped with path planning and intelligent obstacle avoidance. Our modeling process maps the elements in activity diagrams to timed automata and implements the algorithms as C-code functions of the model, such that the model represents the entire control system of AWL. The (T)CTL queries used for verification completely express the informal requirements written in natural language, and provided by industry. The counter-examples that we have found during verification are helpful for the future optimization of the control system and the design of algorithms. Our model is the abstraction of the actual system, which serves to check correctness of the system at design level. Future work includes proving the correctness of the transformation from activity diagrams to UPPAAL TA and automating this process, modeling the dynamics of the AWL, injecting probabilistic events in the model in order to construct and verify a model that is closer to reality, etc.

Acknowledgement: The research leading to the presented results has been undertaken within the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, funded by the Swedish Knowledge Foundation, grant number: 20150022.

Bibliography

[1] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, 3098:87–124, 2004.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal 4.0, 2006.

[3] Michael S Andersen, Rune S Jensen, Thomas Bak, and Michael M Quottrup. Motion planning in multi-robot systems using timed automata. *IFAC Proceedings Volumes*, 37(8):597–602, 2004.

[4] Adina Aniculaesei, Daniel Arnsberger, Falk Howar, and Andreas Rausch. Towards the verification of safety-critical autonomous systems in dynamic environments. *arXiv preprint arXiv:1612.04977*, 2016.

[5] T.J. Koo, R.Q. Li, M.M. Quottrup, C.A. Clifton, R. Izadi-Zamanabadi, and T. Bak. A framework for multi-robot motion planning from temporal logic specifications. *Science China Information Sciences*, 55(7):1675–1692, 2012. cited By 4.

[6] Arash Khabbaz Saberi, Jan Friso Groote, and Sarmen Keshishzadeh. Analysis of path planning algorithms: a formal verification-based approach. In *ECAL*, pages 232–239, 2013.

[7] Steve Rabin. Game programming gems, chapter a* aesthetic optimizations. *Charles River Media*, 2000.

[8] LanAnh Trinh, Mikael Ekström, and Baran Çürüklü. Dipole flow field for dependable path planning of multiple agents. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, September 2017.

[9] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[10] Rajeev Alur and David Dill. The theory of timed automata. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 45–73. Springer, 1991.

[11] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[12] Alex Nash and Sven Koenig. Any-angle path planning. *AI Magazine*, 34(4):85–107, 2013.

[13] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[14] S. L. Smith, J. Tůmová, C. Belta, and D. Rus. Optimal path planning under temporal logic constraints. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3288–3293, Oct 2010.

[15] T John Koo, Rongqing Li, Michael M Quottrup, Charles A Clifton, Roozbeh Izadi-Zamanabadi, and Thomas Bak. A framework for multi-robot motion planning from temporal logic specifications. *Science China Information Sciences*, pages 1–18, 2012.

[16] Michael Melholt Quottrup, Thomas Bak, and RI Zamanabadi. Multi-robot planning: A timed automata approach. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4417–4422. IEEE, 2004.

[17] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2020–2025. IEEE, 2005.

[18] Calin Belta and LCGJM Habets. Constructing decidable hybrid systems with velocity bounds. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 1, pages 467–472. IEEE, 2004.

[19] Suresh Jeyaraman, Antonios Tsourdos, Ratal Zbikowski, and Brian White. Formal techniques for the modelling and validation of a co-operating uav team that uses dubins set for path planning. In *American Control Conference, 2005. Proceedings of the 2005*, pages 4690–4695. IEEE, 2005.

[20] Eman Rabiah and Boumediene Belkhouche. Formal specification, refinement, and implementation of path planning. In *Innovations in Information Technology (IIT), 2016 12th International Conference on*, pages 1–6. IEEE, 2016.

[21] Rim Saddem, Olivier Naud, Karen Godary Dejean, and Didier Crestani. Decomposing the model-checking of mobile robotics actions on a grid. *IFAC-PapersOnLine*, 50(1):11156–11162, 2017.

[22] Samir Chouali, Azzedine Boukerche, and Ahmed Mostefaoui. Ensuring the reliability of an autonomous vehicle: A formal approach based on component interaction protocols. In *Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems*, pages 317–321. ACM, 2017.

[23] Valeriy Vyatkin and Hans-Michael Hanisch. Verification of distributed control systems in intelligent manufacturing. *Journal of Intelligent Manufacturing*, 14(1):123–136, 2003.

[24] Hans-Michael Hanisch, Andrei Lobov, Jose L Martinez Lastra, Reijo Tuokko, and Valeriy Vyatkin. Formal validation of intelligent-automated production systems: towards industrial applications. *International Journal of Manufacturing Technology and Management*, 8(1-3):75–106, 2006.

Chapter 9

Paper B: Towards a Two-Layer Framework for Verifying Autonomous Vehicles

Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist.
Proceedings of the 11th Annual NASA Formal Methods Symposium. Springer, 2019.

Abstract

Autonomous vehicles rely heavily on intelligent algorithms for path planning and collision avoidance, and their functionality and dependability can be ensured through formal verification. To facilitate the verification, it is beneficial to decouple the static high-level planning from the dynamic functions like collision avoidance. In this paper, we propose a conceptual two-layer framework for verifying autonomous vehicles, which consists of a static layer and a dynamic layer. We focus concretely on modeling and verifying the dynamic layer using hybrid automata and UPPAAL SMC, where a continuous movement of the vehicle as well as collision avoidance via a dipole flow field algorithm are considered. In our framework, decoupling is achieved by separating the verification of the vehicle's autonomous path planning from that of the vehicle autonomous operation in its continuous dynamic environment. To simplify the modeling process, we propose a pattern-based design method, where patterns are expressed as hybrid automata. We demonstrate the applicability of the dynamic layer of our framework on an industrial prototype of an autonomous wheel loader.

9.1 Introduction

Autonomous vehicles such as driverless construction equipment bear the promise of increased safety and industrial productivity by automating repetitive tasks and reducing labor costs. These systems are being used in safety- or mission-critical scenarios, which require thorough analysis and verification. Traditional approaches such as simulation and prototype testing are limited in their scope of verifying a system that interacts autonomously with an unpredictable environment that assumes the presence of humans and varying site conditions. These techniques are either applied later in the system's development cycle (testing), or they simply cannot prove, exhaustively or statistically, the satisfaction of properties related to autonomous behaviors such as path planning, path following, and collision avoidance (simulation). Formal verification is usually adopted to compensate such shortage, yet verifying such a complex system in a continuous and dynamic environment is still considered a big challenge [1][2].

In this paper, we approach this challenge by proposing a two-layer framework consisting of a *static* and a *dynamic* layer, which facilitates verifying autonomous vehicles. The structure of the framework separates the static high-level path planning that assumes an environment with a predefined sequence of milestones that need to be reached, as well as static obstacles, from the dynamic functions like collision avoidance, thus providing a separation of concerns for the system's design, modeling, and verification. To improve on existing formal models of vehicle movement [3][4], in the dynamic layer, we propose a continuous model of the vehicle's motion, together with a model of the environment, where moving obstacles are either predefined or dynamically generated. The resulting models are hybrid automata, as accepted by the input language of UPPAAL Statistical Model Checker (SMC). The vehicle's dynamics is modeled as ordinary differential equations assigned to locations in the hybrid automata. In this paper, the hybrid automata only have non-deterministic time-bounded delays that are encoded based on the default uniform distributions assigned by UPPAAL SMC. We also consider the embedded control system of the autonomous vehicle including the involved processes, as well as the scheduling and communication among them. The path planning is following the Theta* algorithm [5], and the collision avoidance relies on the dipole flow field one [6]. Both algorithms are encoded as C-code functions in UPPAAL SMC, within the dynamic layer of our framework. Once this is accomplished, we can statistically model check the resulting network of hybrid automata, against probabilistic invariance properties expressed in weighted metric temporal logic [7]. To simplify the modeling process, we propose a pattern-based design method

to provide reusable templates for various components of the framework. We demonstrate the applicability of our approach for modeling and analyzing the dynamic layer on an industrial autonomous wheel loader prototype that should meet certain safety-critical requirements.

This paper is organized as follows. In Section 9.2, we overview hybrid automata and UPPAAL SMC, as well as the Theta* algorithm for path planning, and the dipole flow field algorithm for collision avoidance. Section 9.3 describes the function of the autonomous wheel loader and its architecture. In Section 9.4, we present the conceptual two-layer framework, and in Section 9.5 we propose the pattern-based modeling of the components (of the dynamic layer) and their formal encoding. Next, we demonstrate the applicability of the framework on the autonomous wheel loader, and we present the verification results in Section 9.6. We compare to related work in Section 9.7, before concluding and outlining future lines of research in Section 9.8.

9.2 Preliminaries

In this section, we overview the background information needed for the rest of the paper, that is, hybrid automata and UPPAAL SMC, as well as the Theta* and dipole flow field algorithms.

9.2.1 Hybrid Automata and UPPAAL SMC

UPPAAL SMC [8] is an extension of the tool UPPAAL[9], which supports statistical model checking of hybrid automata (HA). A HA is defined as the following tuple:

$$HA = < L, l_0, X, \Sigma, E, F, I >, \tag{9.1}$$

where: L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, X is a finite set of continuous variables, $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions that are partitioned into inputs (Σ_i) and outputs (Σ_o), E is a finite set of edges of the form (l, g, a, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$, $a \in \Sigma$ is an action label, and φ is a binary relation on $\mathbb{R}^X$, $F(l)$ is a delay function for the location $l \in L$, and I assigns an invariant predicate $I(l)$ in/of L, which bounds the delay time in the respective location. In UPPAAL SMC, locations are marked as *urgent* (denoted by encircled u) or *committed* (denoted by encircled c), indicating that time cannot progress in such locations. Committed locations are more restrictive, requiring that the next edge to be traversed needs to start from a committed location. The delay function $F(l)$ for a simple clock variable

x, which is used in (priced) timed automata, is encoded as the linear differential equation $x' = 1$ or $x' = e$ appearing in the invariant of l.

The semantics of the HA is defined over a timed transition system, whose states are pairs $(l, u) \in L \times \mathbb{R}^X$, with $u \vDash I(l)$, and transitions defined as: (i) delay transitions ($< l, u > \xrightarrow{d} < l, u + d >$ if $u \vDash I(l)$ and $(u + d') \vDash I(l)$, for $0 \leq d' \leq d$), and (ii) discrete transitions ($< l, u > \xrightarrow{a} < l', u' >$ if edge $l \xrightarrow{g,a,r} l'$ exists such that $a \in \Sigma, u \vDash g$, clock valuation u' in the target state (l', u') is derived from u by resetting all clocks in the reset set r of the edge, such that $u' \vDash I(l')$).

In UPPAAL SMC, the automata have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions, and (ii) the non-deterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or user-defined exponential distributions for unbounded delays. In this paper, only the default uniform distributions for time-bounded delays are used. Moreover, the UPPAAL SMC model is a network of HA that communicate via broadcast channels and global variables. Only broadcast channels are allowed for a clean semantics of purely non-blocking automata, since the participating HA repeatedly race against each other, that is, they independently and stochastically decide on their own how much to delay before delivering the output, with the "winner" being the automaton that chooses the minimum delay.

UPPAAL SMC supports an extension of *weighted metric temporal logic* for probability estimation, whose queries are formulated as follows: `Pr[bound] (ap)`, where `bound` is the simulation time, `ap` is the statement that supports two temporal operators: "$Eventually$" ($\Diamond$) and "$Always$" ($\Box$). Such queries estimate the probability that `ap` is satisfied within the simulation time bound. Hypothesis testing (`Pr[bound]` $(\psi) \geq p_0$) and probability comparison (`Pr[bound]` $(\psi_1) \geq$ `Pr[bound]` (ψ_2)) are also supported.

9.2.2 Theta* Algorithm

In this paper, we employ the Theta* algorithm to generate an initial path for our autonomous wheel loader. The Theta* algorithm has been firstly proposed by Nash et al. [5] to generate smooth paths with few turns, from the starting position to the destination, for a group of autonomous agents. Similar to the A* algorithm that we have used in our previous study [3], the Theta* algorithm explores the map and calculates the cost of nodes by the function $f(n) = g(n) + h(n)$, where n is the current node being explored, $g(n)$ is

the Euclidean distance from the starting node to n, and $h(n)$ is the estimated cheapest cost from n to the destination. In this paper, we use Manhattan distance [10] for $h(n)$. In each search iteration, the node with the lowest cost among the nodes that have been explored is selected, and its reachable neighbors are also explored by calculating their costs. The iteration is eventually ended if the destination is found or all reachable nodes have been explored. As an optimized version of A*, Theta* determines the preceding node of a node to be any node in the searching space instead of only neighbor nodes. In addition, Theta* adds a line-of-sight (LOS) detection to each search iteration to find an any-angle path that is less zigzagged than those generated by A* and its variants. For the detailed description of the algorithm, we refer the reader to the literature [5].

9.2.3 Dipole Flow Field for Collision Avoidance

Searching for a path from the starting point to the goal point, assuming a large map, is not an easy task and it is usually computationally intensive. Hence, some studies have adopted methods to generate a small deviation from the initial path, which is much easier to compute than an entirely new path, while being able to avoid obstacles. To avoid collisions, Trinh et al.[6] propose an approach to calculate the *static flow field* for all objects, and the *dynamic dipole field* for the moving objects in the map. In the theory of dynamic dipole field, every object is assumed to be a source of magnetic dipole field, in which the magnetic moment is aligned with the moving direction, and the magnitude of the magnetic moment is proportional to the velocity. In this approach, the static flow field is created within the neighborhood of the initial path generated by the Theta* algorithm. The flow field force is a combination of the attractive force drawing the autonomous wheel loader to the initial path, and the repulsive force pushing it away from obstacles. Unlike the dipole field force, the flow field force always exists, regardless of whether the vehicle is moving or not. As soon as the vehicle equipped with this algorithm gets close enough to a moving obstacle, the magnetic moment around the objects keeps them away from each other. The combination of the static flow field and the dynamic dipole field ensures that the vehicle moves safely by avoiding all kinds of obstacles and that it eventually reaches the destination, as long as a safe path exists. Compared with other methods [11][12], this algorithm provides a novel method for path planning of mobile agents, in the shared working environment of humans and agents, which suits our requirements well. For details, we refer the reader to the literature [6].

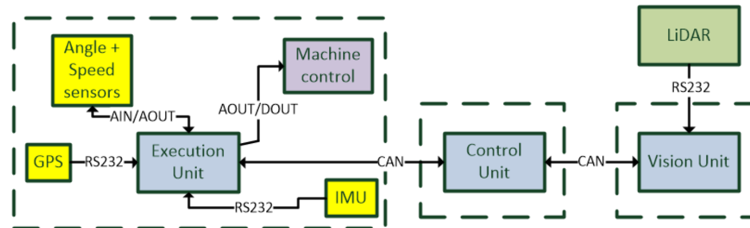


Figure 9.1: The architecture of the AWL's embedded control system

9.3 Use Case: Autonomous Wheel Loader

In this section, we introduce our use case, which is an industrial prototype of an autonomous wheel loader (AWL) that is used in construction sites to perform operations without human intervention [3]. On one hand, like other autonomous vehicles, autonomous wheel loaders need to be equipped with path-planning and collision-avoidance capabilities. On the other hand, they also ought to accomplish several special missions, e.g., autonomous digging, loading and unloading, often in a predefined sequence. Furthermore, autonomous wheel loaders usually work in unpredictable environments – dust and various sunlight conditions (from dim to extremely bright) that might cause inaccuracy or even errors in image recognition and obstacle detection. Moving entities, e.g., humans, animals, and other machines, might also behave unpredictably, for there are no traffic lights and lanes. Despite such disadvantages, the AWL's movements are less restricted if compared to, for instance, self-driving cars, as there are only a few traffic rules in sites. They can also stop and wait as long as they need without influencing the vehicles behind them. All these characteristics make our path-planning (Theta*) and collision-avoidance (Dipole Flow Field) algorithms applicable.

The architecture of the AWL's control system, presented in Figure 9.1, consists of three main units: a vision unit, a control unit, and an execution unit, which are connected by CAN buses. In this paper, we mainly focus on the control unit that consists of three parallel processes, namely `ReadSensor`, `Main`, and `CalculateNewPath`, as depicted in Figure 9.2. These three processes are executed in parallel on independent cores. The process `ReadSensor` acquires data from sensors (e.g., LIDAR, GPS, angle and speed sensors, etc.) and sends them to the shared memory before they are accessed by process `Main` that runs the path-planning algorithm and invokes a function called

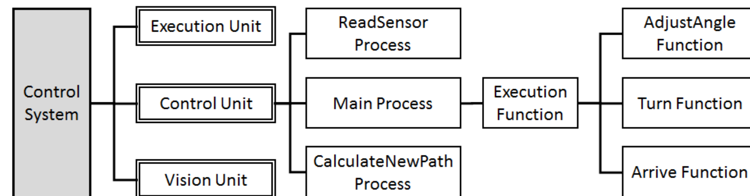


Figure 9.2: Process allocation in the control system

`Execution Function`, in which three sub-functions are called. The function `AdjustAngle` adjusts the moving angle of the AWL, based on its own and the obstacles' positions. Function `Turn` judges if the AWL arrives at one of the milestones on its initial path calculated by the path-planning algorithm, and changes its direction based on the result. Function `Arrive` judges if the AWL reaches the destination and sends the corresponding commands. Basically, the processes `Main` and `ReadSensor` are responsible for the AWL's regular routine. However, when an unforeseen obstacle suddenly appears in its vision, the process `Main` sends a request to process `CalculateNewPath`, in which the collision-avoidance algorithm is executed and a new and safe path segment is generated if it exists. Note that, although the AWL has more functionality, e.g., digging and loading, we focus only on the path planning and collision avoidance in this paper.

The loader's architecture (Figures 9.1, 9.2), including the parallel processes and functions, is hierarchical. Moreover, the distributed nature of the AWL's components, and the dynamic nature of its movement (including collision avoidance) call for a separation of concerns along the static and the dynamic dimensions of the system. Hence, in the following, we propose a two-layer framework to model and verify autonomous vehicles on different levels.

9.4 A Two-level Framework for Planning and Verifying Autonomous Vehicles

As it is shown in Figure 9.3, our two-level framework consists of a static layer and a dynamic layer, between which data is exchanged according to a defined/-chosen communication protocol. The *static layer* is responsible for path and mission planning for the AWL, according to possibly incomplete information

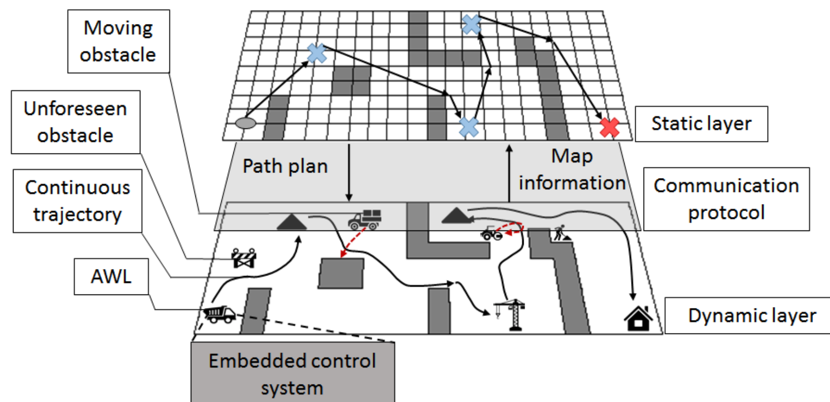


Figure 9.3: Two-layer framework for planning and verifying autonomous vehicles

of the environment. In this layer, known static obstacles are assumed, together with milestones representing points of operation of the loader. The *dynamic layer* is dedicated to simulating and verifying the system following the reference path given by the static layer, while considering continuous dynamics in an environment containing moving and unforeseen obstacles.

Static layer. The static layer is defined as a tuple $< E_s, S_s, M_s >$, where E_s denotes a discrete environment, S_s is a set of known static obstacles, and M_s is a set of milestones associated to missions (e.g., digging, loading, unloading, charging), including the order of execution, and timing requirements. As the path found by the path-planning algorithm is a connection of several straight-line segments on the map, realistic trajectories and continuous dynamics do not need to be considered in this layer. Hence, the environment is modeled as a discrete Cartesian grid whose resolution is defined appropriately to present various sizes of static obstacles, e.g., holes, rocks, signs, etc. Even if not entirely faithful to reality, the Cartesian grid provides a proper abstraction of the map for path and mission planning. As the static layer is still at the conceptual stage currently, we propose several possible options for modeling and verification of this layer. DRONA [13] is a programming framework for building safe robotics systems. which has been applied in collision-free mission planning for drones. Rebeca is a generic tool for actor-based modeling and has been proven to be applicable for motion planning for robots [14]. Mission Management

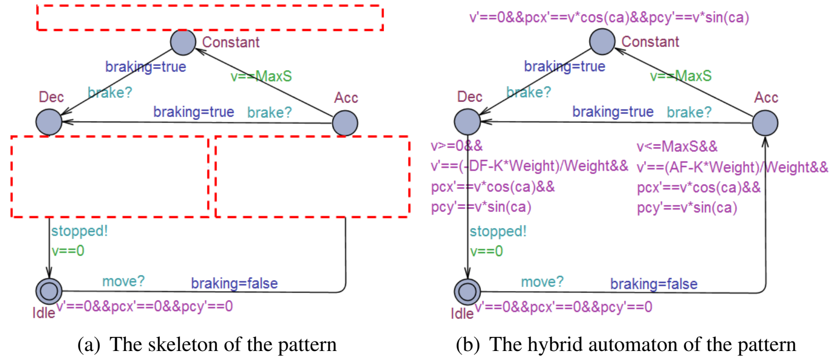
Tool (MMT) is a tool allowing a human operator an intuitive way of creating complex missions for robots with non-overlapping abilities [15].

Dynamic layer. The dynamic layer is defined as a tuple $< E_d, T_s, S_d, M_d, D_d >$, where E_d is a continuous environment, T_s is the trajectory plan input by the static layer, S_d is a set of static obstacles, M_d is a set of moving obstacles that are predefined, D_d is a set of unforeseen moving obstacles that are dynamically generated. The speed and direction of a moving obstacle $m_0 \in M_d$ are predefined as constant values in our model. The dynamically generated moving obstacle $d_0 \in D_d$ is instantiated during the verification when its initial location, moving speed and angle are randomly determined. Collision-avoidance algorithms are executed in this layer if the vehicle meets moving obstacles or unforeseen static obstacles. Ordinary differential equations (ODEs) are adopted to model the continuous dynamics of moving objects (e.g., vehicle, human, etc.), and the embedded control system of the autonomous vehicle is modeled in this layer.

This two-layer design has many benefits. Firstly, it provides a separation of concerns for the system's design, modeling, and verification. As a path plan does not concern the continuous dynamics of the vehicle, the discrete model in the static layer is a proper abstraction, which sacrifices some unnecessary realistic elements but preserves the possibility of exhaustive verification. The dynamic layer, which concerns the actual trajectories of moving objects, consists of hybrid models that contain relatively more realistic details of the system and environment, which enhance the truthfulness of the model. However, as a tradeoff, only probabilistic verification is supported in this layer. In addition, modification of algorithms or design is only restricted within the corresponding layer, so potential errors will not propagate in the entire system. Secondly, the two-layer framework is open for extension. It provides a possibility to add layers for new functions, such as artificial intelligence or centralized control.

9.5 Pattern-based Modeling of the Dynamic Layer

A classic control system consists of four components: a plant containing the physical process that is to be controlled, the environment where the plant operates, the sensors that measure some variables of the plant and the environment, and the controller that determines the system state and outputs timed-based signals to the plant [16]. In our case, as shown in Figure 9.1, the execution unit is the "plant" that describes the continuous dynamics of the AWL. The "sensors" are divided into two classes: vision sensors (LiDAR) connecting to the vision



(a) The skeleton of the pattern (b) The hybrid automaton of the pattern

Figure 9.4: The pattern of the linear motion component in the execution unit

unit, and motion sensors (GPS, IMU, Angle and Speed sensors) connecting to the execution unit.

9.5.1 Patterns for the Execution Unit

Currently, the vision unit and vision sensors have no computation ability, so they are simply modeled as data structures. The execution unit is modeled in terms of hybrid automata, in which the motion of the AWL is given by a system of three ordinary differential equations:

$$\dot{x}(t) = v(t)cos\theta(t) \quad \dot{y}(t) = v(t)sin\theta(t) \tag{9.2}$$

$$\dot{\theta}(t) = \omega(t), \tag{9.3}$$

where, $\dot{x}(t)$ and $\dot{y}(t)$ are the projections of the linear velocity on x and y axes, $\omega(t)$ is the angular velocity, and $v(t)$ is the linear velocity, which follows the Newton's Law of Motion: $v(t) = \frac{F - k \times M}{M}$, where F is the force acting on the AWL, k is the friction coefficient, and M is the mass of the AWL.

The pattern of the execution unit is a hybrid model consisting of two hybrid automata, namely linear motion and rotation. Here we use the linear motion component as an example to present the idea. As depicted in Figure 9.4(a), there are four locations indicating four moving states of the AWL, that is, stop at `Idle`, acceleration at `Acc`, moving at a constant speed at `Constant`, and deceleration at `Dec`. Therefore, the derivatives of the position (pcx', pcy') and the velocity (v') are assigned to zero at `Idle` for the stop state. According to different moving states, variations of equation 9.2 should be encoded in the

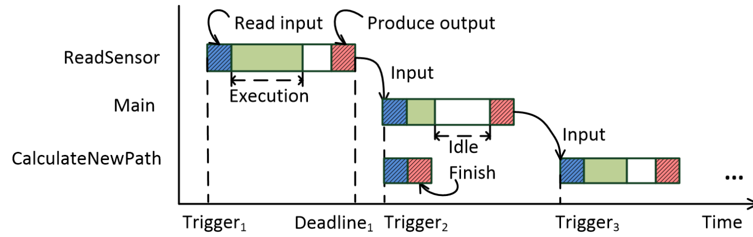


Figure 9.5: Process scheduling

refinement of each location in the blank boxes in 9.4(a). Figure 9.4(b) is an instance of the pattern, where v' is set to a positive value ($v' == (AF - k * m)/m$) at location Acc to present acceleration. Once the velocity reaches the maximum value ($maxS$) or the automaton receives a brake signal (denoted as a channel $brake$), it goes to location Constant or Dec, where the ODEs are changed to make the AWL move at a constant speed or decelerate.

9.5.2 Patterns for the Control Unit

As a part of an embedded system, the control unit model has three basic components: a scheduler, a piece of memory, and a set of processes. Currently, the memory is modeled as a set of global variables, hence the scheduler pattern and the processes patterns are the essence. Due to its safety-critical nature, the control unit is assumed to be a multi-core system and the processes are scheduled in a parallel, predictable, and non-preemptive fashion. This scheduling policy is inspired by *Timed Multitasking* [16], which tackles the real-time programming problem using an event-driven approach. However, instead of the preemptive scheduling, we apply a non-preemptive strategy. To illustrate this scheduling strategy, we use the three processes in the control unit (Figure 9.2) as an example. The process ReadSensor is firstly triggered at the moment $Trigger_1$ when the process reads data from sensors and runs its function as illustrated in Figure 9.5. Regardless of the exact execution time of a process, the inputs are consumed and the outputs are produced at well-defined time instances, namely trigger and deadline. As the input of Main is the output of ReadSensor, the former is triggered after the latter finishes. At same the moment, CalculateNewPath finishes its execution immediately as no input comes. This is actually reasonable, since process CalculateNewPath does not need to be executed every round, as it is responsible for generating a new path segment only when the AWL encounters an obstacle. For the bene-

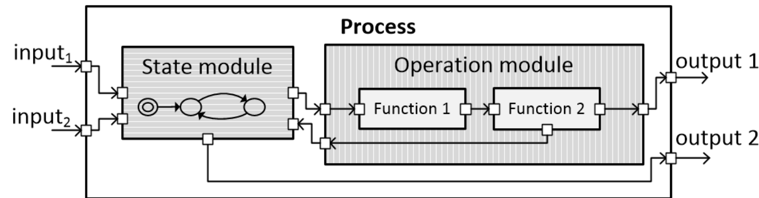


Figure 9.6: A process model example

fits brought by the explicit execution time and deadline, we refer the interested readers to the literature [16] for detail.

The pattern of a process consists of two parts: a state module and an operation module. Similar to the state machine function-block and modal function-block in related work [17], the state module describes the mode transition structure of the processes, and the operation module describes the procedure or computation of the process. Because of their definition, the state modules are modeled as discrete automata, and the operation modules are modeled as discrete automata or computation formulas according to their specific functionality. Figure 9.6 shows the inputs of the process coming to the state module in which the state of the process transfers according to the inputs. Some state transitions of the state module are detailed by the functions in the operation module in the sense that the former invokes the latter for concrete computation. Specifically, functions in the operation module could be modeled as discrete automata when they involve logic, or executable code when they are purely about computation. After executing the corresponding functions in the operation module, some results are sent out of the process as output, and some are sent back to the state module for state transitions, which might also produce output. The designs of the state module and operation module for different processes have both similarities and differences. They all need to be scheduled, to receive input, produce output, etc., but their specific functionality is different. To make our patterns reusable, we design fixed skeletons of the process patterns, which are presented as hybrid automata.

9.5.3 Encoding the Control Unit Patterns as Hybrid Automata

Scheduler. To model the scheduler as a hybrid automaton in UPPAAL SMC, we first discretize the continuous time as a set of basic time units to mimic the clock in an embedded system. As depicted in Figure 9.7, we use an invariant at location `Init` (clock `xd` $\leq$ `UNIT`), and a guard on its outgoing edge (`xd` $==$

UNIT) to capture the coming basic time unit. We also declare a data structure representing processes, as follows:

```
typedef struct{
   int id; //process id
   bool running; // whether the process is being executed
   int period; //counter for the period of the process
   int executionTime; //counter for the execution time of the
       process
}PROCESS;
```

When a basic time unit comes, the scheduler transfers to location Updating. In the function update(), the period counters of all processes are decreased by one, and so are the execution time counters if the variable running in the process structure is true. When the period of a process equals zero, its id is inserted into a queue called ready and the variable readyLen indicating the length of the queue is increased by one. Similarly, when the executionTime equals zero, the process's id is inserted into a queue called done. The fact that the queue done is not empty (doneLen > 0) implies that the execution times of some processes have elapsed, so the scheduler changes from Updating to Finishing to generate the outputs of those processes. The self loop at location Finishing indicates that the outputs of all the processes in queue done are generated orderly by the synchronization between the scheduler and the corresponding process automaton via the channel output. If the queue ready is not empty (readyLen > 0), similarly, the scheduler moves to location Execution to trigger the top process in ready via the channel execute, and waits there until the process finishes, when the scheduler is then synchronized again with the process via channel finish. Note that the process finishes its function instantaneously and stores its output in the local variables, which will only be transferred to the other processes via global variables when the execution time passes.

Process. A typical state module of a process consists of four states: being triggered, doing its own function, idle, and output. A typical pattern for it is shown in Figure 9.8(a). Except locations Start and Idle, all locations are urgent because the execution is instantaneous, and the output is generated when the execution time is finished. From location Start to O1, the process is being triggered by the scheduler by synchronizing on channel execute[id], in which id is the process's ID. If the input is valid (input $==$ true), the process starts to execute by leaving O1 to the next location, otherwise, it finishes its execution immediately by going back to Start without any output generated, just as the description of the scheduling policy in Section 9.5.2. The

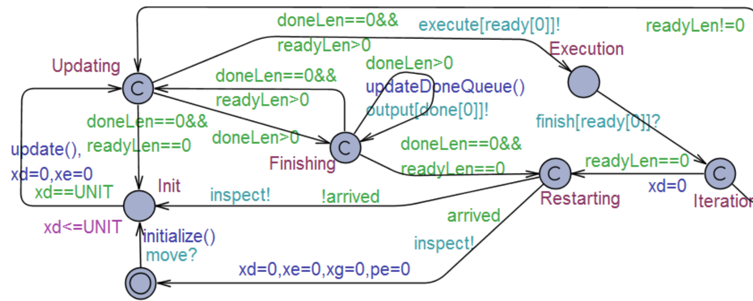
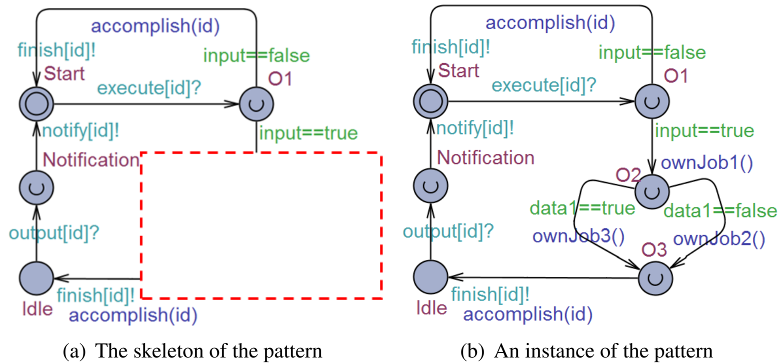


Figure 9.7: The pattern of the scheduler

blank box indicates the process's own function that is created in an ad-hoc fashion, so it is not part of the fixed skeleton of the pattern. After executing its own function, the process synchronizes again with the scheduler on channel `finish[id]`, when the process finishes and gives control back to the scheduler. The output is generated from location `Idle` to `Notification`. The broadcast channel `notify[id]` is for notifying other processes waiting for the output of the current process. Based on this idea, we give an example instantiated from this pattern in Figure 9.8(b). The automaton goes from `O2` to `O3` through two possible edges based on `data1`, which is the outcome of function `ownJob1()`. The concrete computation is encoded in functions `ownJob2()` and `ownJob3()`, which are the counterparts of the functions in the operation module of Figure 9.6. If the specific function of the process is more complex than in this example, or it includes function invocation, this blank box can be extended with synchronizations with other automata. We will elaborate this by revisiting our use case in the next section.

9.6 Use Case Revisited: Applying Our Method on AWL

As the patterns of linear motion and rotation components and the scheduler are totally applicable in the use case, they are simply transplanted in the model of the AWL with parameter configuration. Hence, in this section, we mainly demonstrate how the processes in AWL's control unit are modeled using the proposed patterns, and present the verification results.



(a) The skeleton of the pattern (b) An instance of the pattern

Figure 9.8: The pattern of a generic process

9.6.1 Formal Model of the Control Unit

The control unit contains three parallel processes (Figure 9.2). `ReadSensor` and `CalculateNewPath` are relatively simple because they do not invoke other functions, while `Main` calls function `Execution`, which calls other three functions: `AdjustAngle`, `Turn`, and `Arrive`. Therefore, The state modules of `ReadSensor` and `CalculateNewPath` are modeled as single automata and the operation modules are the functions at edges encoding the computation of their functionality. Differently, the state module of `Main` is a mutation of the process pattern extended with a preprocessing step calculating an initial path by running Theta* algorithm. Figure 9.9 depicts the automaton of the state module of `Main`, in which another automaton representing the function `Execution` is invoked via channel `invoke[0]`, where 0 is the ID of the function `Execution`. Note that the transition from the location `Init` to `Moving` is the preprocessing step and Theta* algorithm is implemented in the function `main`, which will be moved to the static layer eventually after the entire framework is accomplished. As the process `Main` invokes other functions, its operation module is a network of automata containing the function `Execution`, `AdjustAngle`, `Turn`, and `Arrive`, which are called by using synchronizations between the state module automata and operation module automata (channels `invoke`, `respond`, `finish`). After calling other functions, `Main` goes to the location `Idle` via three edges based on the return values of the invoked functions and waits to generate output there.

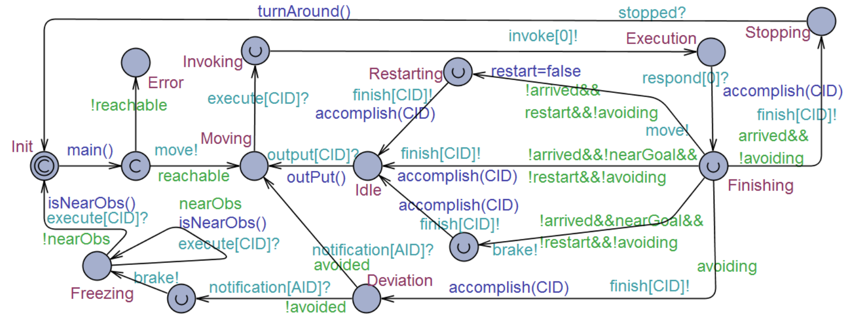


Figure 9.9: The automaton of the state module of the process `Main`

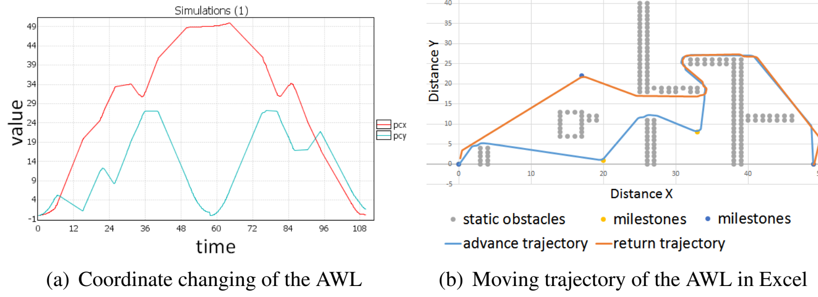
9.6.2 Statistical Model Checking of the AWL Formal Model

Environment configuration. In the following we consider a continuous map with the size 55×55, where five static obstacles and two moving obstacles are predefined, and another moving obstacle is dynamically generated during the verification. In order to achieve this, we leverage the spawning command of UPPAAL SMC to instantiate new time automata instance of the moving obstacle that “appears” in the map whenever it is generated by the automaton called `generator` and “disappears” from the map when its existence time terminates. The speed of the moving obstacles is a constant value indicating that they move one unit distance per second and their moving directions are either opposite or the same as it of the AWL. The parameters of the AWL are the weight of it, acceleration and deceleration force, friction coefficient and maximum speed, which are defined as constant values in UPPAAL SMC.

Path generation and following. Given a start and a goal and a set of milestones, the AWL must be able to calculate a safe path passing through them orderly avoiding static obstacles if the path exists and follow it. To verify this requirement, we first simulate the model in UPPAAL SMC using the command:

$$simulate\ 1[<= 110]\ \{pcx, pcy\} \tag{9.4}$$

where `pcx` and `pcy` are the real-valued coordinate of the AWL. Figure 9.10(a) shows the result of the simulation, and the result data is exported into Excel to depict the moving trajectory of the AWL shown in Figure 9.10(b). The AWL perfectly follows the generated path that avoids all the static obstacles. But the simulation only runs one possible execution trace of the AWL model. Hence,



(a) Coordinate changing of the AWL (b) Moving trajectory of the AWL in Excel

Figure 9.10: Moving trajectory of the AWL generated by the command {`simulate 1[<=110] pcx,pcy`} in UPPAAL SMC and exported in Excel

we further verify the model with a query:

$$Pr[<= 70](<> \ arrived \ \&\& \ counter <= 60) \tag{9.5}$$

$$Pr[<= 110]([] \ followedPath) \tag{9.6}$$

where `arrived` and `counter` in query 9.5 are a Boolean variable and a clock that reflect if the AWL arrives at the destination and what the minimum time does it take, `followedPath` in query 9.6 is a Boolean variable indicating if the AWL has reached the destination and come back to the start by visiting all the milestones orderly. To update the value of `followedPath` timely and periodically during the verification, we create an independent automaton called `monitor` that checks the index of the model. The `monitor` is triggered by the `scheduler` every time unit that is small enough to ensure the position of the AWL does not change much during this time interval. The probability interval of satisfying these queries is [0.902606, 1] with 95% confidence obtained from 36 runs.

Collision avoidance. By the nature of the Theta* algorithm, AWL is able to avoid the static obstacles as long as it sticks to the initial path. When it meets an unforeseen static obstacle or a moving obstacle, the AWL must run the dipole flow field algorithm timely to avoid it. Two queries are designed to get the simulated moving trajectory and estimate the probability of satisfaction:

$$simulate \ 1[<= 110] \ \{pcx, pcy, ocx[0], ocy[0], ocx[1], ocy[1], ocx[3], ocy[3]\} \tag{9.7}$$

$$Pr[<= 110]([] \ !collided) \tag{9.8}$$

Arrays ocx and ocy in query 9.7 represent the positions of moving obstacles at x and y axes. The trajectories got from query 9.7 is shown in Figure 9.11, where “A” and “B” are two predefined moving obstacles and “C” is a dynamically generated obstacle that moves “recklessly” towards the AWL, so the latter turns around to avoid the obstacle. The overlap of two trajectories at “C” does

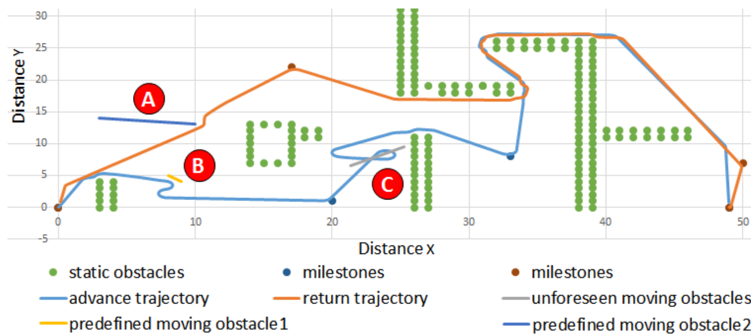


Figure 9.11: The trajectory of the AWL in a map with three moving obstacles

not imply a collision because the AWL and the moving obstacle are not at the same position at the same moment. To prove this, query 9.8 is designed, where collided is a Boolean variable indicating if the AWL has collided with any static or moving obstacles during the verification time. Similar to the verification of path generation and following, the automaton monitor is extended to update this variable periodically by checking if the current coordinate of the AWL is close to any obstacle in the map, and the threshold of the distance is 0.8 in this case. The probability interval of satisfying this query is [0.902606,1] with 95% confidence obtained from 36 runs.

9.7 Related Work

Automata-based methods [18][19][4][20] have been used for path or motion planning. Different from our work, these studies aim to solve the vehicle-routing problem by using temporal logic. These studies accomplish many typical autonomous tasks like searching for an object, avoiding an obstacle, and missions sequencing. However, as they focus on achieving collision avoidance in design, uncertainties in the real deployment like transmission time of

sensors data in the embedded system and unforeseen obstacles have not been considered.

Runtime verification that monitors the behavior of autonomous systems complements this shortage to some extend [21][22][23][24]. This technique extracts information from a running system, based on which the behavior of the system is verified. Runtime overhead caused by the monitor is the most common problem introduced by this method.

Agent-based method is another widely studied approach for autonomous systems [25][26][21][27][28]. As the predominant form of rational agent architecture is that provided through the Beliefs, Desires, and Intentions (BDI) approach, these studies aim to translate the agent-based language to a formal language to verify the behavior of the agent. But this method usually does not concern the detail of the embedded control system and continuous dynamics of the vehicle.

There are also some studies providing a framework for verification of autonomous vehicles or robots. In [29], the authors captured the behavior of an unmanned aerial vehicle performing cooperative search mission into a Kripke model to verify it against the temporal properties expressed in Computation Tree Logic (CTL). Their model contains a decision making layer and a path planing layer. In [30], the authors propose an approach combining model checking with runtime verification to bridge the gap between software verification (discrete) and the actual execution of the software on a real robotic platform in the physical world. The software stack of a robotics system providing different verification capability focusing on different functionality has inspired our work. However, our framework provides an ability to encode the collision avoidance algorithm in the model and verifying it in a continuous environment.

9.8 Conclusions and future work

We have proposed a conceptual two-layer framework for formally verifying autonomous vehicles that decouples the high-level static planning from dynamic functions like collision avoidance, etc. The framework provides a separation of concerns for the complex modeling and verification of autonomous vehicles. The static layer focuses on making the optimal plan for the vehicle to accomplish a sequence of missions based on the incomplete information of the environment. While the dynamic layer concerns the execution of the plan with vehicle dynamics in a continuous environment model where unforeseen

moving obstacles appear randomly. Hence, a collision avoidance algorithm relying on dipole flow field is implemented in the model of the embedded control system in this layer. We are currently engaged in modeling the dynamic layer using hybrid automata and UPPAAL SMC, and designing a pattern-based method to simplify the modeling process and increase reusability. The dynamic layer has been applied to model and verify a prototype of an autonomous wheel loader and the verification result shows the capability and applicability of statistical model checking adopted in autonomous vehicles. We expect to report our research of the static layer and the combination of these two layers in the years to come.

Acknowledgment The research leading to the presented results has been performed within the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, funded by grant 20150022 of the Swedish Knowledge Foundation that is gratefully acknowledged.

Bibliography

[1] Amit Bhatia, Matthew R Maly, Lydia E Kavraki, and Moshe Y Vardi. Motion planning with complex goals. *IEEE Robotics & Automation Magazine*, 18(3):55–64, 2011.

[2] Michael S Branicky, Vivek S Borkar, and Sanjoy K Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE transactions on automatic control*, 43(1):31–45, 1998.

[3] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Formal verification of an autonomous wheel loader by model checking. In *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, pages 74–83. ACM, 2018.

[4] Michael Melholt Quottrup, Thomas Bak, and RI Zamanabadi. Multi-robot planning: A timed automata approach. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 4417–4422. IEEE, 2004.

[5] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[6] Lan Anh Trinh, Mikael Ekström, and Baran Cürüklü. Toward shared working space of human and robotic agents through dipole flow field for dependable path planning. *Frontiers in neurorobotics*, 12, 2018.

[7] Peter Bulychev, Alexandre David, Kim Guldstrand Larsen, Axel Legay, Guangyuan Li, Danny Bøgsted Poulsen, and Amelie Stainer. Monitor-based statistical model checking for weighted metric temporal logic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 168–182. Springer, 2012.

[8] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. *arXiv preprint arXiv:1208.3856*, 2012.

[9] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[10] Paul E Black. Manhattan distance. *Dictionary of Algorithms and Data Structures*, 18:2012, 2006.

[11] Luis Valbuena and Herbert G Tanner. Hybrid potential field based control of differential drive mobile robots. *Journal of intelligent & robotic systems*, 68(3-4):307–322, 2012.

[12] Yoav Golan, Shmil Edelman, Amir Shapiro, and Elon Rimon. Online robot navigation using continuously updated artificial temperature gradients. *IEEE Robotics and Automation Letters*, 2(3):1280–1287, 2017.

[13] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. Drona: A framework for safe distributed mobile robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pages 239–248. ACM, 2017.

[14] Ali Jafari, Jayasoorya Jayanthi Surendran Nair, Stephan Baumgart, and Marjan Sirjani. Safe and efficient fleet operation for autonomous machines: an actor-based approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 423–426. ACM, 2018.

[15] Branko Miloradović, Baran Cürüklü, Mikael Ekström, and Alessandro Papadopoulos. Extended colored traveling salesperson for modeling multi-agent mission planning problems. In *Proceedings of the 8th International Conference on Operations Research and Enterprise Systems - Volume 1: ICORES,*, pages 237–244. INSTICC, SciTePress, 2019.

[16] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[17] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. Comdes-ii: A component-based framework for generative development of distributed real-time control systems. In *null*, pages 199–208. IEEE, 2007.

[18] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2020–2025. IEEE, 2005.

[19] Marius Kloetzer and Cristian Mahulea. A petri net based approach for multi-robot path planning. *Discrete Event Dynamic Systems*, 24(4):417–445, 2014.

[20] Stephen L Smith, Jana Tumova, Calin Belta, and Daniela Rus. Optimal path planning for surveillance with temporal-logic constraints. *The International Journal of Robotics Research*, 30(14):1695–1708, 2011.

[21] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[22] Erann Gat, Marc G Slack, David P Miller, and R James Firby. Path planning and execution monitoring for a planetary rover. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 20–25, 1990.

[23] Alex Lotz, Andreas Steck, and Christian Schlegel. Runtime monitoring of robotics software components: Increasing robustness of service robotic systems. In *Advanced Robotics (ICAR), 2011 15th International Conference on*, pages 285–290. IEEE, 2011.

[24] Chenxia Luo, Rui Wang, Yu Jiang, Kang Yang, Yong Guan, Xiaojuan Li, and Zhiping Shi. Runtime verification of robots collision avoidance case study. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 204–212. IEEE, 2018.

[25] Rafael H Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Autonomous agents and multi-agent systems*, 12(2):239–256, 2006.

[26] 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.

[27] Michael Fisher, Rafael H Bordini, Benjamin Hirsch, and Paolo Torroni. Computational logics and agents: a road map of current technologies and future trends. *Computational Intelligence*, 23(1):61–91, 2007.

[28] Michael Fisher, Louise Dennis, and Matt Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, 2013.

[29] Gopinadh Sirigineedi, Antonios Tsourdos, Brian A White, and Rafal Zbikowski. Modelling and verification of multiple uav mission using smv. *arXiv preprint arXiv:1003.0381*, 2010.

[30] Ankush Desai, Tommaso Dreossi, and Sanjit A Seshia. Combining model checking and runtime verification for safe robotics. In *International Conference on Runtime Verification*, pages 172–189. Springer, 2017.

Chapter 10

Paper C: TAMAA: UPPAAL-based Mission Planning for Autonomous Agents

Rong Gu, Eduard Enoiu, and Cristina Seceleanu.
Proceedings of the 35th ACM/SIGAPP Symposium On Applied Computing (SAC). ACM, 2020.

Abstract

Autonomous vehicles, such as construction machines, operate in hazardous environments, while being required to function at high productivity. To meet both safety and productivity, planning obstacle-avoiding routes in an efficient and effective manner is of primary importance, especially when relying on autonomous vehicles to safely perform their missions. This work explores the use of model checking for the automatic generation of mission plans for autonomous vehicles, which are guaranteed to meet certain functional and extra-functional requirements, e.g., timing ones. We propose a model of autonomous vehicles as agents in timed automata together with monitors for supervising their behavior in time, for instance battery level. We automate this approach by implementing it in a tool called TAMAA (Timed-Automata-based Planner for Multiple Autonomous Agents) and integrating it with a mission-management tool. We demonstrate the applicability of our approach on an industrial autonomous wheel loader use case.

10.1 Introduction

Autonomous vehicles [1] are complex systems that combine mechanical elements, electromechanical devices, digital circuits and software programs in embedded controllers. Their operation is subjected to many constraints, due to cluttered environments and objectives that can change over time. Mission planning is the process of determining what each autonomous vehicle should do to achieve the goals of the mission as described by the high-level system specifications. This includes autonomous path planning such that (static) obstacles are avoided, tasks assignment and scheduling, and re-planning in unforeseen circumstances. The challenge in this area is the development of modeling and verification frameworks [2, 3] able to accommodate the operating complexity of these systems, while allowing for the verification of their designs early in the development process. One way of ensuring the quality of mission design for autonomous vehicles is to employ model checking for generating mission plans with guaranteed correctness. In this study, we propose such an approach to synthesize mission plans for autonomous agents, and apply our approach on a use case provided by Volvo Construction Equipment (VCE), which involves autonomous wheel loaders (AWL) working in quarries. To facilitate understanding, let us assume that our agents are AWL, which are in fact involved in the industrial use case that we apply our approach on. An AWL is designed to move and function autonomously and fulfil complex requirements, for instance, *"Dig stones at the stone pile. Carry and unload them into a primary crusher 500 meters away. Avoid obstacles and keep repeating these tasks until the stone pile is empty or the AWL needs to charge."* In order to specify such requirements rigorously, synthesize mission plans for autonomous agents, and verify their execution formally, we adopt the two-layer framework approach for modeling and verifying autonomous agents, proposed in our previous work [4]. Our framework builds on the established principle of separation of concerns, and consists of a *static layer* and a *dynamic layer*. The *static layer* focuses on path planning and task scheduling and the *dynamic layer* focuses on modeling the kinematics of the agents to verify if they can accomplish the tasks and circumvent risks, such as moving or unforeseen obstacles, while executing the mission plans. The contribution of this paper targets only the design of the *static layer* of such a framework, which provides rigorous algorithms for model generation and a user-friendly tool for model configuration.

Specifically, we use Timed Automata [5] and Timed Computation Tree Logic (TCTL) [6] for capturing formally the autonomous agents' behavior and requirements specification, respectively. Formal definitions of the concepts,

e.g., tasks, are given to support the model-generation algorithms that are created and integrated with an advanced path-planning algorithm (Theta* [7]) to generate formal models. For simplicity, we use mission plans to denote path-and-task plans in this paper. The formal models are built for synthesizing mission plans that satisfy requirements like the one we aforementioned. These requirements often concern three aspects: i) *safety*: all obstacles of the generated paths should be avoided, ii) *execution constraints*: tasks should be executed with respect to given logical and temporal constraints, iii) *timeliness*: the final goal should be achieved within a certain amount of time for productivity reasons. Given such a mixed palette of requirements, it is not trivial to generate automatically mission plans that guarantee all of them. Most related work proposes solutions for synthesizing mission plans without deadline constraints, but just respecting a required ordering of tasks. Moreover, it is desirable that such synthesis of plans is supported by an easy-to-use tool, in which the user can visualize and modify the mission plans.

Hence, in this work, we propose a method supported by a tool, called TAMAA (Timed-Automata-based Planner for Multiple Autonomous Agents). Our approach integrates the state-of-the-art model checker UPPAAL [8], suited for verifying real-time systems, with a toolkit for mission configuration called MMT (Mission Management Tool) [9]. TAMAA implements the model-generation algorithms and provides a graphic interface to configure the environment, agents, and tasks and organizes the information to build formal models, including the movement of agents, task execution, and monitors. Next, within TAMAA, one can verify the generated model with UPPAAL, against the TCTL queries that formalize the natural-language requirements and generate diagnostic traces. The traces are parsed by TAMAA to synthesize mission plans. Eventually, the synthesis result is shown in MMT. If there is a valid path, it is guaranteed to be correct and optimal in the sense that it is generated via exhaustive model checking. If no valid path exists, a counter-example is depicted to illustrate the contradictions in the model configuration. We demonstrate the applicability and assess of TAMAA by applying it to scenarios of an industrial use case.

The novelty of TAMAA is that it addresses not only path generation but also takes into account complex requirements (e.g., timing ones). Moreover, our solution combines a rigorous, formal encoding of algorithms for computation with a user-friendly interface for visualizing model configurations. Model-generation algorithms provide an automatic way for obtaining formal models, which is less time-consuming and error-prone than the manual generation alternative.

The remainder of the paper is organized as follows. In Section 10.2 we

introduce the preliminaries of this paper. Section 10.3 describes the actual contribution, that is, TAMAA, whereas in Section 10.4 we introduce the implementation and evaluation of TAMAA. In Section 10.5 we compare to related work, before concluding and outlining possible future work in Section 10.6.

10.2 Preliminaries

10.2.1 UPPAAL Timed Automata

A *timed automaton* (TA) is an extended finite-state automaton suitable for modeling real-time systems [10]. UPPAAL [6] is a tool for modeling, simulation, and model checking of real-time systems, and uses an extension of TA as the modeling formalism. A UPPAAL timed automaton is defined as a tuple: $< L, l_0, A, V, C, E, I >$, where L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $A = \Sigma \bigcup \tau$ is a set of *actions*, where Σ is a finite set of *synchronizing actions* and $\tau \notin \Sigma$ are internal actions, V is a set of *data variables*, C is a set of real-valued variables called *clocks*, $E \subseteq L \times B(C, V) \times A \times 2^C \times L$ is the set of *edges*, where $B(C, V)$ is the set of *guards* over C and V, that is, conjunctive formulas of clock constraints $B(C)$ (of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C, n \in \mathbb{N}, \bowtie \in \{<, \leq, =, \geq, >\}$) and non-clock constraints $B(V)$, and $I : L \mapsto B_{dc}(C)$ is a function assigning *invariants* to locations where $B_{dc}(C) \subseteq B(C)$ denotes a subset of clock constraints resulting from the restriction to upper bounds $\triangleleft \in \{<, \leq\}$.

The semantics of a TA is given by a *labeled transition system*. The states of the labeled transition system are pairs (l, u), where $l \in L$ is the current location, and $u \in R^C_{\geq 0}$ is the clock valuation in location l. The initial state is denoted by (l_0, u_0), where $\forall x \in C, u_0(x) = 0$. Let $u \models g$ denote that clock value u satisfies guard g. We use $u + d$ to denote the time elapse where all the clock values have increased by d, for $d \in \mathbb{R}_{\geq 0}$. The following transitions ($\rightarrow$) can happen in a timed automaton:

- Delay transitions: $< l, u > \xrightarrow{d} < l, u + d >$ if $u \models I(l)$ and $(u + d') \models I(l)$, for $0 \leq d' \leq d$, and
- Action transitions: $< l, u > \xrightarrow{a} < l', u' >$ if $l \xrightarrow{g,a,r} l', a \in \Sigma, u \models g$, clock valuation u' in the target state (l', u') is derived from u by resetting all clocks in the reset set r of the edge, such that $u' \models I(l')$.

A network of TA, $B_0 \parallel ... \parallel B_{n-1}$, is a parallel composition of n TA over C, A and synchronization channels (i.e., $a!$ is synchronized with $a?$ by handshake). We refer the reader to the literature [10] for more information on

the theory of TA.

UPPAAL uses a decidable subset of (Timed) Computation Tree Logic [6] as the query language. It consists of path formulae and state formulae. Specifically, we use the following path-specific temporal operators: “Always" ($\Box$) temporal operator for which a given formula is true in all states of a path, and the “Eventually" ($\Diamond$) operator used to show that a formula becomes true in finite time, in some state along a path. In this paper, we use queries of the following categories: (i) *Invariance* (i.e., $A \Box p$), stating that p should be true in all reachable states for all paths, and (ii) *Reachability* (i.e, $E \Diamond p$), stating that there exists a path starting at the initial state, such that p is eventually satisfied along that path.

10.3 TAMAA Approach

In this section, we describe an approach to automatically synthesize correctness-guaranteed mission plans for autonomous agents. We first describe the function and architecture of an industrial use case, the AWL, in Section 10.3.1, which motivates and supports the design of TAMAA, even though our approach is intended to be generic. Next, in Section 10.3.1, we introduce the components and workflow of TAMAA, followed by formal definitions of autonomous agents, their movement, tasks and their execution, which are all needed for the automatic model generation. Last, we describe the model-generation algorithms in Section 10.3.4.

10.3.1 Use Case: Autonomous Wheel Loader

In this section, we introduce our use case, which is based on an industrial system provided by VCE in Sweden. The use case contains Autonomous Wheel Loaders (AWL) that are used in construction sites to perform operations without human intervention. For example, as shown in Figure 10.1, we consider the case of AWL that are utilized to dig and transport stones in a quarry. The AWL first digs a given stone pile before it carries the stones to the primary crusher. Thereafter, the AWL loads the crushed stones and continues moving to the secondary crusher to unload the stones and completes one round of work. During this process, the AWL performs its tasks autonomously and moves to the charging point when its battery level is low. The AWL must avoid obstacles, e.g, holes on the ground. The problem involves mission planning, path following, and collision avoidance.

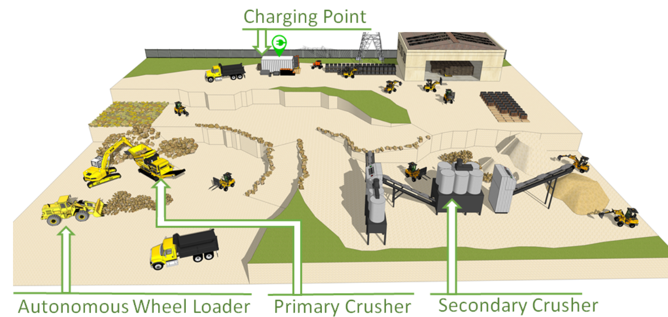


Figure 10.1: An example of a working environment for an autonomous wheel loader

In this paper, we focus on generating valid paths for autonomous agents, guaranteed to avoid static obstacles, as well as correct schedules for the operational tasks of the agents. We assume a two-layer approach of the design, as proposed in our previous work [4], with mission planning belonging to the static layer, whereas the avoidance of dynamic obstacles, including the case of overlapping paths of multiple agents moving at the same time, is being dealt with in the dynamic layer. We assume that the latter functions correctly, and we focus only on synthesizing mission plans.

We apply this generic approach on the AWL use case. Specifically, the requirements of AWL from industry can be divided into the following categories:

- *Task Coverage*. Each AWL must execute all tasks and repeat them until the ultimate goal is achieved (e.g., all stones are transferred to the secondary crusher).
- *Task Matching*. Each AWL must perform certain tasks at particular milestones (e.g., digging stones at the stone pile).
- *Task Sequencing*. The task execution order must be correct.
- *Timing*. Each AWL must finish the tasks within a prescribed time, to keep the desired productivity (e.g., an AWL must complete carrying a ton of stones in 0.5 hours).
- *Event Reaction*. Some special tasks are only triggered by events under certain circumstances. For instance, when the battery level is below a certain level, the AWL must move to the charging point to charge itself.

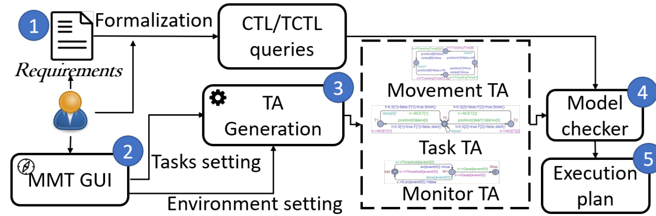


Figure 10.2: Overview of the process of model generation and mission plan synthesis in TAMAA

Overall Challenge. *Given an environment containing one or several AWL with accurate speed control and a deterministic speed range, predefined milestones and static obstacles, and a set of requirements (e.g., task coverage, task matching and sequencing, timing, and reacting to events), synthesize mission plans for these AWL in this environment, such that the requirements are satisfied.*

In the following, we introduce our TAMAA approach for the automatic synthesis of correct mission plans for autonomous agents.

10.3.2 Workflow of TAMAA

Given this challenge, we propose a method called TAMAA (Timed-Automata-based Planner for Multiple Autonomous Agents) for computing an optimal plan for the autonomous agents to accomplish a sequence of tasks based on a set of given requirements. Overall, the approach is composed of the steps shown in Figure 10.2: i) Step 1 - formalizing the requirements into CTL/TCTL queries, ii) Step 2 - configuring the information of the environment and tasks in MMT, iii) Step 3 - automatically generating the UPPAAL TA of movement, tasks, and monitors, iv) Step 4 - verifying models generated in Step 3 in UPPAAL against the queries of Step 1, and generating execution traces that satisfy or violate the queries, and v) Step 5 - using the traces to obtain the mission plans in cases when the requirements are met, or counter-examples when no mission plan exists in the environment configuration. Since this is an automatic approach, users are only involved in the first two steps in the configuration phase of Figure 2. All steps are described in detail in the following sections.

10.3.3 Model Formalization and Definitions of Concepts

In this section we define formally the elements of TAMAA, that is: autonomous agents and their movement and tasks. To illustrate the formal definitions and algorithms, we use a running example extracted from our use case.

Running example. *As depicted in Figure 10.3(a), an AWL starts from A, goes to the stone pile at B and digs stones and moves to the crusher at C to unload stones and comes back eventually.*

An AWL can be considered as an autonomous agent that is situated within an environment, can sense the environment and act on it, over time, in pursuit of its own goals [11]. In this paper we focus on mission planning of autonomous agents, whose movement and tasks are simply abstracted as time durations without considering any real-time feedback from the environment. Therefore, autonomous agents can be considered automated agents at this level of abstraction and defined as follows: *An automated agent is a system that receives instructions from its mission plan and executes its instructions with no human control and no interaction with its environment*. There are many definitions of automated agents in different fields of research [11]. In this paper we assume the definition above, and we formalize an automated agent as follows:

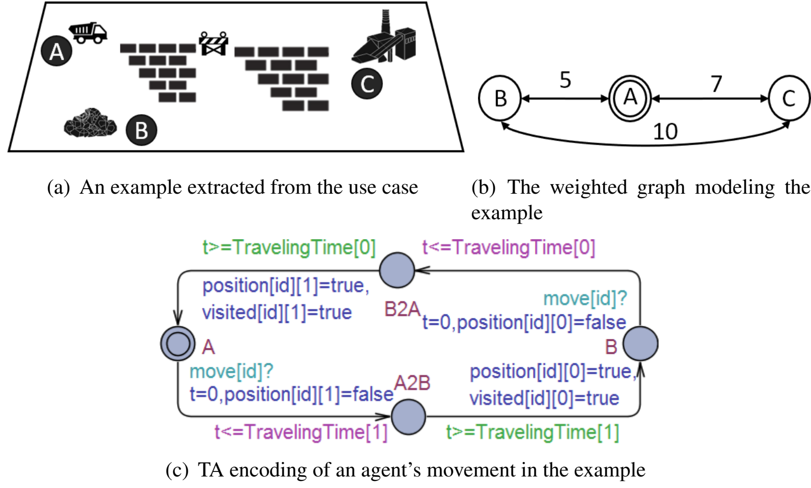
Definition 1 (Automated Agent). *An automated agent (AA) is defined as a tuple:*

$$AA \triangleq < \mathcal{S}, \mathcal{M}, \mathcal{T} >, \tag{10.1}$$

where:

- *$\mathcal{S}$ is the speed of the moving agent,*
- *$\mathcal{M}$ is a set of motion primitives that make the agent move and execute tasks,*
- *$\mathcal{T}$ is a set of tasks that the agent has to accomplish.* □

The working environment of an agent is a closed space including some static obstacles that the agent should avoid, and some milestones where the tasks should be carried out. However, when an agent is reaching a milestone, it does not necessarily stop. According to the mission plan, the agent can stop and execute the corresponding task or simply pass. Static obstacles and milestones are represented as a set of X-Y coordinates in the environment. The working environment of an agent is defined as a weighted graph $G = (V_g, E_g)$, where V_g is a set of vertices denoting the milestones, $E_g \subseteq V_g \times N_g \times V_g$ is a set



(a) An example extracted from the use case (b) The weighted graph modeling the example

(c) TA encoding of an agent's movement in the example

Figure 10.3: A running example and its corresponding weighted graph and TA

of edges, where $N_g \subseteq \mathbb{R}_{\geq 0}$ denotes a set of traveling times between vertices. Edges only connect the vertices that are directly reachable from each other, which means the shortest path between two connected vertices does not pass any other vertices.

We assume that automated agents are equipped with a set of motion primitives that allow them to deterministically move from v to v' for each $(v, t, v') \in E_g$, with $v, v' \in V_g$. Hence, the traveling time *t* between two vertices is constant, and it is calculated by graph-search algorithms such as Theta* algorithm [7]. The weighted graph extracted from the example of Figure 10.3(a) is depicted in Figure 10.3(b).

When agents start to move, their positions depend on the connectivity of vertices and the traveling time. Hence, the movement of agents involves discrete changes of position and the continuous evolution of time, which makes TA a suitable formalism for modeling.

Definition 2 (Movement of AA). *The movement of an AA is defined as a timed automaton in a restricted form:*

$$Mm \triangleq < P, p_0, x_m, A_m, E_m, I_m >, \quad (10.2)$$

where:

- $P = P_v \cup P_e$ *is a finite set of locations, where* P_v *denotes the vertices of*

the weighted graph of the $environment$, and P_e denotes the locations of components encode moving between vertices. The component consists of an incoming edge, a location in P_e and an outgoing edge;

- $p_0 \in P_v$ *is the initial location denoting the initial position;*
- x_m *is a clock variable defined to measure the traveling time;*
- $A_m = \{move\} \cup \tau$ *is a set of actions, where "$move$" is for synchronizing with the automaton encoding the agent's tasks, and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization;*
- $E_m \subseteq P_v \times A_m \times B_m(x_m) \times 2^C \times P_e$ *is a set of edges, where $B_m(x_m)$ is a set of guards containing clock constraints of the form $x_m \geq \Upsilon$, where $\Upsilon \in \mathbb{R}_{\geq 0}$ is a constant value of the traveling time between two locations, and $C = \{x_m\}$;*
- $I_m : P_e \mapsto B_e(x_m)$ *is a function that assigns invariants to locations in P_e, where $B_e(x_m)$ contains clock constraints of the form $x_m \leq \Upsilon$.* □

Based on Definition 2, a part of the TA that models the movement of agents is depicted in Figure 10.3(c), where the agent moves from A to B and and vice versa. Locations A2B and B2A, belonging to P_e, and the their associated invariants are created to model the duration of traveling.

Any automated agent should carry out tasks that can be operations (e.g., digging for AWL) or simply a state of stop. A task is only allowed to be executed at certain predefined positions, with an execution time given as an interval. For example, an AWL only unloads rocks at a primary crusher or a secondary crusher. Some tasks, like charging, are triggered in special circumstances, but once they are triggered they must be prioritized. Given an agent $(\mathcal{S}, \mathcal{M}, \mathcal{T})$ and a set of events Ev triggering $T_i \in \mathcal{T}$, one needs to formally capture the agent's tasks and their execution, which we introduce by the following Definitions 3 and 4, respectively.

Definition 3 (AA Task). *A task is defined as a tuple:*

$$Task \triangleq (B, W, \Delta, S, F, V, G, R, O, M), \tag{10.3}$$

where:

- *B is the best case execution time,*
- *W is the worst case execution time,*

- Δ *is the time that has elapsed during the execution of a task,*
- *S is a Boolean variable denoting if the task has started,*
- *F is a Boolean variable denoting if the task has finished,*
- *V is a set of variables that are changed after the task finishes,*
- *G is a set of Boolean variables (events) that trigger the task,*
- *R is a precondition that must be met before the task starts,*
- *O is a postcondition that must be met after the task finishes,*
- *M is a set of indices of milestones where the task is allowed to be executed.* □

To simplify the notation, T_i is used to denote a task for any $i \in \mathbb{N}$ and we use "." to access an element in the tuple. For instance, $T_i.\Delta$ is designed to measure the total execution time of a task, so $B \leq \Delta \leq W$. The precondition is $T_i.R = \theta_t(T_0.F, ..., T_k.F) \wedge \theta_e(ev_0, ..., ev_m)$, where θ_t and θ_e are predicates reflecting the execution order of tasks $\{T_0, ..., T_k\} \subseteq \mathcal{T} \setminus T_i$, and the status of events $\{ev_0, ..., ev_m\} = Ev$. The postcondition is $T_i.O = \bigwedge_{i=1}^{n} \neg ev_i \wedge T_i.F$, where $ev_i \in T_i.G$ and $n = |T_i.G|$.

There are three tasks $\{T_1, T_2, T_3\}$ in the example of Figure 10.3(a), namely digging the stone pile, loading, and unloading, respectively. The rules of execution are: T_3 can start after T_1 and T_2 finish, T_2 can start after T_1 finishes, ev_0 triggers T_1; then the preconditions and postconditions of these tasks are:

$$T_1.R = ev_0,\ T_1.O = \neg ev_0 \wedge T_1.f$$

$$T_2.R = T_1.f \wedge \neg ev_0,\ T_2.O = T_2.f$$

$$T_3.R = T_1.f \wedge T_2.f \wedge \neg ev_0,\ T_3.O = T_3.f$$

For some tasks, e.g., digging stones, finishing an execution means a decrease of the volume of the stone pile. This feature is reflected in the value change of the variables in $T_i.V$. When an agent is executing a regular task, it must not move. After finishing tasks, the agent must switch to a special task called no-op task before it starts to move. The no-op task indicates that no task is being executed, and it is denoted by $T_0(0, \infty_+, \Delta, S, F, \emptyset, \emptyset, M, \emptyset, \emptyset)$. In T_0, B is 0 and W is ∞_+, implying the execution time can be any length, R and O

are "$\emptyset$" implying the agent can get to or out of this task without restrictions, M is the complete set of all the milestones in the $environment$ implying that this task is allowed at any position (except obstacles), and V and G are $\emptyset$ implying that this task does not change any data variable and is not triggered by any event. Based on Definition 3, we define the execution of tasks of an automated agent AA, as follows.

Definition 4 (Task Execution of AA). *For an automated agent ($\mathcal{S}, \mathcal{M}, \mathcal{T}$), the execution of tasks in $\mathcal{T}$ is defined as a timed automaton in a restricted form:*

$$Taa \triangleq (N, l_0, x_e, A_e, V_e, E_e, I_e, M_e), \tag{10.4}$$

where:

- N *is a set of locations representing the tasks in* $\mathcal{T}$,
- $l_0 \in N$ *is the initial location representing the no-op task* T_0,
- x_e *is a clock that is reset whenever a task finishes,*
- $A_e = \{move,\ done_0, ..., done_n\} \cup \tau$ *is a set of actions,*
- V_e *is a set of variables containing variables of all the tasks in* $\mathcal{T}$, *i.e.,* $V_e = \bigcup_{i=1}^{S} T_i.V$, $S = |\mathcal{T}|$,
- $E_e \subseteq l_0 \times A_e \times B_e(x_e, \mathcal{T}) \times 2^C \times 2^{\mathcal{T}} \times N$ *is a set of edges connecting* l_0 *and* $l \in N$ *with a set of actions and guards, where* $C = \{x_e\}$,
- $I_e : N \setminus l_0 \mapsto B_i(x_e)$ *is a function assigning invariants to locations except* l_0,
- $M_e : N \mapsto \mathcal{T}$ *is a function assigning tasks to locations.* □

In A_e, "$move$" and "$done_0, ..., done_n$" are used for the synchronization between the *task TA* and the *movement TA* and the *monitor TA* respectively. The *monitor TA* are for supervising some indices of the agents that we will introduce later. For $\forall e_i \in E_e$, they are always between l_0 and $l \in N$, because the agents have to switch to the no-op task before they move or execute the next task. For $\forall\, T_i \in \mathcal{T} \setminus T_0$, the invariant $B_i(x_e)$ is of the form $x_e \leq T_i.W$. The guard on the incoming edge of T_i is of the form $P_j \wedge T_i.R$, where the Boolean variable P_j denotes if the current position of the agent is a milestone $m_j \in T_i.M$. The guard on the outgoing edge of T_i is $x_e \geq T_i.B$. Clock x_e, variables $v_i \in T_i.V$, task flags $T_i.S$ and $T_i.F$ are updated on the corresponding edge, respectively.

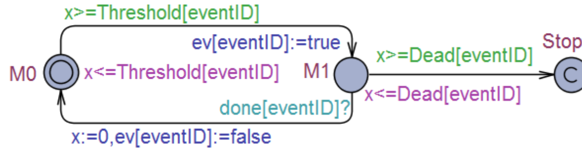


Figure 10.4: A monitor as an UPPAAL TA

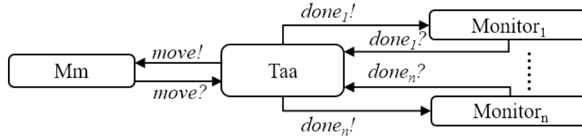


Figure 10.5: A network of TA obtained by TAMAA

As some tasks are triggered by events (e.g., when the battery level is below a certain level, a battery-low event occurs and it triggers the agent to charge), we need monitors to supervise the status of the agents and inform in a real-time manner when the values of the indices are below or above thresholds. In fact, these indices are rates of consumption that could be represented by real-number values of time, e.g., the fuel/electricity at 0.8 rate when the agent travels a certain period of time. In this paper, we assume that all events concern only the indices changing monotonically and continuously over time. An example of monitor TA is depicted in Figure 10.4. The invariant of $M0$ and the guard of its outgoing edge are used to guarantee that the monitor is progressing to $M1$ when the clock's value reaches the threshold. The invariant of M1, and guard of its outgoing edge are used when switching to $Stop$ when the clock's value reaches a certain threshold, meaning that the agent has no resources to move anymore. Hence, the monitor gives the agent a time horizon between the threshold and deadline to react to the event. However, if the agent ignores it for too long, energy (fuel, battery, etc.) is consumed so the former cannot move, which is represented as a deadlock in the TA.

A network of TA $Mm \mid\mid Taa \mid\mid Monitor_1 \mid\mid ... \mid\mid Monitor_n$ over (A, X) is a composition of TA for the movement, tasks, and monitors (Figure 10.5), where $A = \{move,\ done_1,\ ...\ , done_n\}$, $X = \{Mm.x_m, Taa.x_e, Monitor_1.x_r,\ ...\ ,$ $Monitor_n.x_r\}$, $n = |Ev|$. Taa sends out synchronization signals $move$ to inform the movement TA that it is allowed to move, and $done_i$ to $Monitor_i$ informing the monitor TA that the task reacting on event ev_i has been carried

out.

10.3.4 Automatic Generation of Autonomous Mission Models via TAMAA

In this section, we describe the algorithms used for generating the corresponding TA, and we also show how we formalize the requirements as UPPAAL CTL/TCTL queries.

Generation of TA modeling the Movement of AA.

To abstract the continuous-space environment as discrete models (as shown in Definition 2 of Section 10.3.3), we decompose the environment into a set of regions. There are two types of decompositions that have been investigated previously in the literature [12, 13]. The *geometry-ignoring* decomposition [12] concerns only a set of regions of interest and ignores the actual geometry of these regions. In contrast, the *geometry-using* decomposition [13] divides the environment by using different types of geometries, like rectangles, triangles, or convex polygons. Both approaches to environment decomposition ensure that propositions are well preserved by the discrete model of the environment and are therefore called *proposition-preserving decompositions* [12].

Our approach combines these two approaches, by dividing the environment into square cells for path calculation between milestones, and abstracting the environment as a TA where milestones and transitions among them are represented. The concrete description is shown in Algorithm 2. We first decompose the environment as a Cartesian grid and abstract the set of milestones as a two-dimensional array (i.e., ms in Algorithm 2) for storing the coordinates. An array tt of integers is used for storing the traveling time between milestones (lines 2 - 5). The Theta* algorithm is used to generate paths and traveling times (lines 6 to 8). In addition, we traverse the elements in ms and create a location (A) in ta for each of these elements (lines 9 and 10). "*MAX*" is the maximum value of integers. After selecting another location (B) in ta other than A, we connect them via a new location C (lines 11 - 15). We use a function "*CreateConnection*" for assigning guards and channels to edges as shown in Definition 4 of Section 10.3.3 (lines 19 - 23). Once the agent moves to a milestone, the corresponding element in the array *position* flips to true, being turned to false when the agent leaves the milestone. Similarly, the array *visit* is used for storing the visited milestones (lines 21 and 24).

Algorithm 2: TA Generation of the Movement of an AA

```
1  Function CreateTA(Environment env)
2      grid :=new CartesianGrid(env)
3      ms :=new Milestones(env)
4      ta :=new TimedAutomata()
5      int tt[][] :=new int[grid.size][grid.size]
6      for m_i ∈ ms do
7          for m_j ∈ ms ∧ m_i ≠ m_j do
8              tt[m_i][m_j] :=ThetaStar(grid,m_i, m_j)
9          end
10     end
11     while ms ≠ ∅ do
12         Select a m_i ∈ ms, create a location A in ta representing it
13         for B ∈ ms ∧ B ≠ A ∧ tt[A][B] < MAX do
14             Create a location C in ta
15             Label C with a guard: ta.c ≤ tt[A][B]
16             CreateConnection(A, C, B, ta)
17             CreateConnection(B, C, A, ta)
18         end
19         Remove m_i from ms
20     end
21     return ta
22 Function CreateConnection(L_1, T, L_2, ta)
23     Create an edge e in ta from L_1 to T
24     Label e with a channel move?
25     Label e with assignments: ta.c := 0, position[L_1] := false
26     Create an edge e′ in ta from T to L_2
27     Label e′ with a guard: ta.c ≥ tt[L_1][L_2]
28     Label e′ with assignments: position[L_2] := true,
         visited[L_2] := true
```

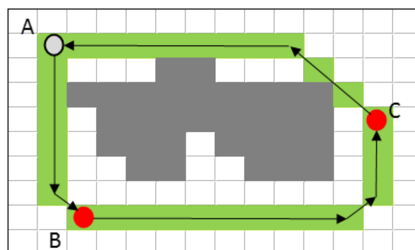


Figure 10.6: Environment decomposition and paths calculation for the environment of Figure 10.3(a)

Figure 10.6 illustrates the decomposition of the environment of Figure 10.3(a). The cells in the Cartesian grid are the decomposition unit. The ones that are completely or partially occupied by obstacles are marked as forbidden cells (colored in grey in Figure 10.6). Consequently, this is a conservative approach for obstacle detection that leads to unnecessary avoidance. This can be solved by increasing the grid resolution, which might however increase the computation time.

Generation of the Task TA for Automated Agents.

Based on the concepts defined in Definition 4 of Section 10.3.3, we describe the process of building task TA (i.e., Algorithm 3).

Note that, the line numbers mentioned in this paragraph refer to Algorithm 3. We first create a TA and an initial location l_0 to represent the no-op task and label the self-loop edge of l_0 with a channel "$move$" for synchronization with the movement TA. In addition, we traverse every task $T_i \in AA.\mathcal{T}$ and create a location l_i in ta to represent it (lines 6 and 7). The l_i edge is labeled with an invariant ta.c $\leq T_i.W$, which ensures that the execution of the task must not be longer than its worst-case execution time. We create a new edge connecting l_0 to l_i and label it with a guard and assignments (lines 9, 10, and 11). The edge denotes the start of T_i and the guard is used to model that the agent must be at one of the locations whose index belongs to $T_i.M$ and that the task's precondition $T_i.R$ must be true (line 10). The assignment on the edge resets the clock and flips the *starting* flag to true and the *finishing* flag to false (line 11). We create an edge connecting l_i to l_0 (lines 12 to 17) and label it with a guard, a channel, and assignments. The tasks triggered by events are labeled with "$done[i]$" to inform the monitor TA that the events are responded to (lines 13 - 15). For all tasks, the assignments reset the clock and update the

Algorithm 3: Task Automaton Generation

```
1  Function CreateTaskAutomaton(Agent aa, Bool position[], EventSet Ev)
2      ta :=new TimedAutomata()
3      Create an initial location l_0 in ta representing the no-op task
4      Create a self-loop edge of l_0 and label it with move!
5      while aa.𝒯 ≠ ∅ do
6          Select a task T_i ∈ aa.𝒯
7          Create a location l_i in ta representing T_i
8          Label l_i with an invariant: ta.c ≤ T_i.W
9          Create an edge e connecting l_0 to l_i
10         Label e with a guard: ⋁_{j=k}^{m} position[j] ∧ T_i.R, where
            {k, ..., m} = T_i.M
11         Label e with assignments: ta.c := 0, T_i.S := true,
            T_i.F := false
12         Create an edge e′ connecting l_i to l_0
13         for ev_i ∈ Ev do
14             if ev_i triggers T_i then
15                 Label e′ with a channel: done[i]!
16             end
17         end
18         Label e′ with assignments: ta.c := 0, T_i.S := false,
            T_i.F := true
19         Label e′ with a guard: ta.c ≥ T_i.B
20         Delete T_i from aa.𝒯
21     end
22     return ta
```

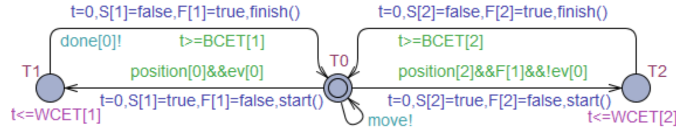


Figure 10.7: A UPPAAL TA for execution of tasks

starting and finishing flags (line 16). For exemplification, we show in Figure 10.7 a task TA, which models the execution of a subset of tasks in our running example, namely digging holes (i.e., T_1) and loading stones (i.e., T_2). T_0 is the no-op task.

Composition of TA.

A network of *movement* TA, *task* TA, and *monitor* TA is constructed to synthesize mission plans satisfying various properties. As shown in Figures 10.3(c) and 10.7, the *task* TA and the *movement* TA are synchronized using the "$move$" channel. Specifically, in the task TA in Figure 10.7, this channel is only labeled in the self-loop of T_0, because the agent is only allowed to move when it has no operation to perform. We mention here that the agent does not necessarily move to milestone C (i.e., $position[2]$) for executing the corresponding task T_2. It probably simply passes it to go to another milestone. Therefore the transition from T_0 to T_2 is not synchonized with the movement TA. The synchronizations between the *task* TA and *monitor* TA are modeled in a similar way. The network of TA is then used for model checking it against certain CTL/TCTL queries in UPPAAL. The resulting execution traces from model checking representing transitions between milestones and tasks are then used to synthesize mission plans.

UPPAAL Queries Design.

We use the requirements in our use case provided by VCE (as described in Section 10.3.1) to show the design of UPPAAL queries in the following way:

- *Task Coverage*. Given that the agent must finish all tasks, the corresponding CTL query can be written as:

$$E\Diamond\ (F[1] \wedge F[2] \wedge ... \wedge F[j] \wedge \textit{stonePileVol} == 0) \qquad (10.5)$$

As shown in Definition 3 in Section 10.3.3, "$F[i]$" represents $T_i.F$ and *stonePileVol* is a variable that belongs to $T_i.V$ and indicates the volume

of the stone pile. Hence, this query requires the agents to finish all the tasks and repeat them. When the query is verified in UPPAAL, a diagnostic trace is generated for the synthesis of mission plans. We make use of UPPAAL's ability to generate traces witnessing submitted reachability properties. Currently, UPPAAL supports three options for trace generation: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.

- *Task Matching*. For this requirement, the agent must execute certain tasks at certain milestones. Assuming that task T_i is allowed at milestone P_i, the corresponding CTL query has the following form:

$$A\Box\ (\texttt{Taa}.T_i\ imply\ \texttt{Mm}.P_i) \tag{10.6}$$

- *Task Sequencing*. This requirement specifies that the order of task execution must be correct. Assuming that task T_i must be done before T_{i+1} starts, one can design the query in the following form:

$$E\Diamond\ S[i+1] \tag{10.7}$$

$$A\Box\ S[i+1]\ imply\ F[i] \tag{10.8}$$

- *Timing Requirement*. For this requirement, tasks must be completed within a time limit. Assume the agent must finish all tasks to carry all stones within N time units, and c is a clock variable, the TCTL query can be as follows:

$$E\Diamond\ (F[1] \wedge ... \wedge F[j] \wedge \mathit{stonePileVol} == 0 \wedge\ c \leq N) \tag{10.9}$$

- *Event Reaction*. For this requirement, special tasks that are triggered by events under some circumstances need to be executed and prioritized. For instance, for battery level checking, a monitor would activate an event when the battery level is lower than a threshold. As the agent model describes all possible combinations of behavior, it is possible that the agent keeps staying at one location or moves meaninglessly without executing any task until its battery is consumed. Nevertheless, the satisfaction of query (10.5) or (10.9) guarantees that the synthesized mission plan subsumes that the agent charges itself whenever the low-battery event occurs, since if a deadlock happens in the monitor TA, there is no way to finish all the tasks (i.e., queries (10.5) and (10.9) cannot be satisfied).

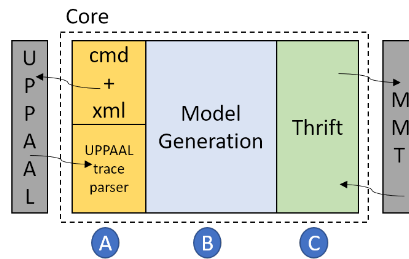


Figure 10.8: The architecture of TAMAA

10.4 TAMAA Implementation and Evaluation

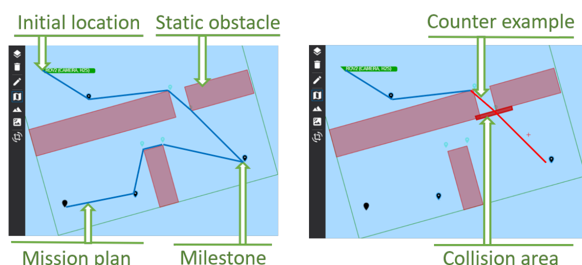
In this section we outline the main aspects of TAMAA, including a high-level implementation description and an evaluation of its applicability and scalability.

10.4.1 Implementation and User Interface

We present several technical solutions used in the implementation of TAMAA to fully support the complexity required for model-checking the generated models. We have implemented the algorithms described in Section 10.3.4 in Java and have integrated the TAMAA tool with a Mission Management Tool (MMT) [1]. MMT is a tool allowing users to graphically create complex environment and missions for agents [9]. One can drag and drop markers in the environment as milestones and assign specific tasks to them (See Figure 10.9). When the environment and tasks are configured in MMT, one can choose a planner from the interface to calculate a mission plan. Our TAMAA tool is linked to MMT as an explicit planner option, which runs the Theta* algorithm, generates the UPPAAL TA and calculates mission plans satisfying the given queries.

As illustrated in Figure 10.8, TAMAA has two communication modules and one processing module, which are implemented in Java. The communication module connecting to UPPAAL is shown as module A in Figure 10.8, and consists of two sub-modules: one for connecting to UPPAAL for model checking and the other one for analyzing the obtained trace. The module for automatic TA generation is module B. The communication module connecting to MMT is shown as module C. Module B first reads data from MMT via module C,

[1]One can find the introduction video on: http://www.es.mdh.se/staff/3552-Rong_Gu



(a) A mission plan generated in a feasible environment

(b) A counter example generated in an unfeasible environment

Figure 10.9: Two screenshots of the MMT user interface

which is implemented in Apache Thrift[2], to obtain information about the environment, agents and tasks. Next, TAMAA executes its model-generation engine for automatically creating the UPPAAL TA, represented as xml files. Module A invokes UPPAAL and sends the generated model and the necessary commands as command-line arguments. After UPPAAL finishes the verification of the model, an execution trace is produced and parsed by module A so that module B can interpret it as a mission plan and transfer it to MMT via module C. Finally, if a satisfactory execution trace exists, the corresponding mission plan is depicted in MMT as it is shown in Figure 10.9(a). Otherwise, a counter example representing an invalid mission plan is also produced and shown in MMT's GUI (See Figure 10.9(b)) for further debugging.

10.4.2 Evaluation of TAMAA's Applicability

In this section, we consider various scenarios of AWL to show the applicability of this method in a realistic setting. The following environment is used in all scenarios: a 50×50 2D space containing 3 static obstacles and 4 milestones. The evaluation is conducted on a machine running an Intel Core i5 processor with 16 GB of RAM and a 64-bit Windows OS. We present here the scenarios and the evaluation results.

Scenario 1. An AWL needs to perform three tasks in the right order for one round. We design queries in the form of queries (10.5)–(10.9) to obtain execution traces and check if the model satisfies the requirements. As all queries are

[2]Apache Thrift is a software framework for scalable cross-language services development. https://thrift.apache.org

Table 10.1: Scalability evaluation results with different number of milestones and tasks, and 1 agent.

Query	Numer of Milstones	Numer of Tasks	Numer of Explored States	Time
Reachability	30	30	20,363	0.2 s
	60	60	157,033	2.2 s
	100	100	712,721	14 s
Invariance	30	30	41,193	0.3 s
	60	60	317,703	4.5 s
	100	100	1,429,903	29 s

satisfied, a mission plan is synthesized within a few milliseconds.

Scenario 2. An AWL needs to repetitively execute four tasks until the stone pile is empty and travel to a certain location to charge itself when the battery is low. In this case, one more task (i.e., charging) is added being triggered by the “low-battery” event. A monitor containing an auxiliary data variable is designed to inform the task TA when the battery level decreases under a certain threshold. A query in the form of query (10.9) is verified and the computation takes 0.5 seconds while exploring 113,719 states. The generated trace for this query shows that the AWL as specified in the model reacts to the event “low-battery” in time.

Scenario 3. In this case, three AWL cooperate to accomplish one complex task. They have to all gather at one milestone and start some task simultaneously. After that, they continue to finish their own tasks. In this situation, the synthesis of mission plans for three agents has to be conducted in one entire model. Similarly, queries in the form given by formulas (10.5)–(10.9) are checked and satisfied. Verifying invariance queries in the form of query (10.8) takes less than 9 s exploring more than 770,000 states. Overall, our results show that mission plans are successfully synthesized for all scenarios within a few seconds. This is an indication that the TAMAA approach is applicable to the industrial scenarios of AWL.

10.4.3 Evaluation of TAMAA’s Scalability

In this section, we consider the scalability of TAMAA with regard to the number of milestones, tasks and agents considered. In all scenarios we are interested in two types of queries: reachability and invariance (queries (10.5) and (10.8) are

Table 10.2: Scalability evaluation results with different number of agents running 3 tasks among 3 milestones.

Query	Numer of Agents	Numer of Explored States	Time
Reachability	2	1,661	0.01 s
	3	159,632	2.0 s
	4	2,058,132	20160 s
	5	Out of Memory	Out of Memory
Invariance	2	3,533	0.03 s
	3	344,701	4.0 s
	4	Out of Memory	Out of Memory

used as examples). In this evaluation, we first vary the number of milestones and tasks between 30 and 100 for both. Meanwhile, we use one AWL for all variations as this is one of the scenarios of the use case. The result is presented in Table 10.1, and it shows that the computation time ranges between 0.2 s and 29 s and the number of explored states is increasing quickly with the number of milestones and tasks for all queries. We mention here that even for a model containing 100 milestones and tasks, the results are encouraging in terms of model checking efficiency.

In addition, we evaluate the scalability of TAMAA by varying the number of AWL between 2 and 5. The results are shown in Table 10.2. The environment is kept the same for all variations and contains three milestones and three tasks. We conduct this evaluation using Scenario 3 described in Section 10.4.2, as it is the most complex one. The number of explored states and computation time increase exponentially with the number of AWL. We observe that the results for the case with three AWL running in a 3-milestone environment executing 3 tasks are similar with the results for one agent executing in an environment with 60 milestones and 60 tasks shown in Table 10.1. This can be explained by the increase in the number of TA and clocks for the models containing more agents, which results in more time zones and non-deterministic interleaving transitions. Thus, searching through models with more agents takes significantly longer. We note that the use of more than three agents is problematic and therefore the method restricts the handling of larger systems, due to the increased cost of computation time and number of explored states. Because of the use of clocks at locations and edges, partial order reduction [14] of the model is not suitable in this model. One of our ongoing work is to integrate reinforcement learning [15] in the model to leverage the historical exploration

of the state space of the model to alleviate the scalability problem. We leave it to report in our future work.

10.5 Related Work

In recent decades, there has been a growing interest in formal modeling and verification of autonomous systems given mission planning problems with complex goals. Belta et al. [2] present a hierarchical structure and based on a three-level process they propose a method using Linear Temporal Logic (LTL). This is evaluated in several case studies [16, 17]. Bhatia et al. [18, 12] propose a multi-layered synergistic approach for solving motion planning problems for mobile robots involving temporal goals. This approach addresses two key issues: the construction of the discrete abstraction of the robots and its efficient exploration in the high-level layer. Dimarogonas et al. [19, 20] propose their method for motion planning of multiple-agent systems using various temporal logic. In contrast to these studies, our approach is focusing on integrating a state-of-the-art path-planning algorithm with temporal logic to leverage the heuristics and efficiency of the former and the rigorousness and expressiveness of the latter. In addition, our approach combines a model-checker with a mission-management tool to tackle this problem on an industrial case, which demonstrates the applicability of this approach in realistic scenarios. Instead of using LTL (e.g., [2]) for requirement specification, we explore the use of TCTL for expressing different types of requirements like timing requirements.

10.6 Conclusions and Future Work

In this paper, we have presented an integrated approach (named TAMAA) for automatically generating mission plans for autonomous agents satisfying various requirements, such as functional and timing ones. As part of TAMAA, we provide formal definitions of the movement of autonomous agents and tasks. These definitions enable the formalization of a practical problem. We also provide algorithms for the automatic model generation before verifying the models in UPPAAL against CTL/TCTL queries expressing requirements of autonomous vehicles. For increasing the appeal of our method, we have implemented these algorithms in a tool and integrated it with a mission-management tool to provide an easy-to-use automated support. Our approach has been evaluated in three scenarios proposed by industry demonstrating its applicability in realistic scenarios. The scalability evaluation shows that while the number

of tasks and milestones do not significantly influence the cost of model checking, the synthesis efficiency dramatically decreases when the number of agents increases.

The future work has at least two potential directions. One is to combine model checking techniques with machine learning (e.g. reinforcement learning) to improve the efficiency of searching through the state space. Another direction is related to the integration of TAMAA with our two-layer framework with the goal of proposing and evaluating an entire solution for performing static planning and dynamic simulation and verification by taking into account the kinematics of different types of agents.

Acknowledgment The research leading to the presented results has been performed within the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, funded by grant 20150022 of the Swedish Knowledge Foundation that is gratefully acknowledged.

Bibliography

[1] Saeed Asadi Bagloee, Madjid Tavana, Mohsen Asadi, and Tracey Oliver. Autonomous vehicles: challenges, opportunities, and future implications for transportation policies. *Journal of modern transportation*, 24(4):284–303, 2016.

[2] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George J Pappas. Symbolic planning and control of robot motion [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):61–70, 2007.

[3] Scott Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1):6, 2017.

[4] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Towards a two-layer framework for verifying autonomous vehicles. In *NASA Formal Methods Symposium*, pages 186–203. Springer, 2019.

[5] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[6] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, 3098:87–124, 2004.

[7] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39, 2010.

[8] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal 4.0. *Department of computer science, Aalborg university*, 2006.

[9] Branko Miloradović, Baran Cürüklü, Mikael Ekström, and Alessandro Papadopoulos. Extended colored traveling salesperson for modeling multi-agent mission planning problems. In *International Conference on Operations Research and Enterprise Systems*, pages 237–244. INSTICC, SciTePress, 2019.

[10] R. Alur and D. Dill. Automata for Modeling Real-time Systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990.

[11] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer, 1996.

[12] Amit Bhatia, Matthew R Maly, Lydia E Kavraki, and Moshe Y Vardi. Motion planning with complex goals. *IEEE Robotics & Automation Magazine*, 18(3), 2011.

[13] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 25(6):1370–1381, 2009.

[14] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *International Conference on Concurrency Theory*, pages 485–500. Springer, 1998.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[16] Stephen L Smith, Jana Tumova, Calin Belta, and Daniela Rus. Optimal path planning for surveillance with temporal-logic constraints. *International Journal of Robotics Research*, 30(14):1695–1708, 2011.

[17] Alphan Ulusoy, Stephen L Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. Optimality and robustness in multi-robot path planning with temporal logic constraints. *International Journal of Robotics Research*, 32(8):889–911, 2013.

[18] Amit Bhatia, Lydia E Kavraki, and Moshe Y Vardi. Sampling-based motion planning with temporal goals. In *International Conference on Robotics and Automation*, pages 2689–2696. IEEE, 2010.

[19] Fernando S Barbosa, Lars Lindemann, Dimos V Dimarogonas, and Jana Tumova. Integrated motion planning and control under metric interval temporal logic specifications. In *2019 18th European Control Conference (ECC)*. IEEE, 2019.

[20] Alexandros Nikou, Dimitris Boskos, Jana Tumova, and Dimos V Dimarogonas. On the timed temporal logic planning of coupled multi-agent systems. *Automatica*, 97:339–345, 2018.

Chapter 11

Paper D: Combining Model Checking and Reinforcement Learning for Scalable Mission Planning of Autonomous Agents

Rong Gu, Eduard Enoiu, Cristina Seceleanu, Kristina Lundqvist.

Technical Report: MDH-MRTC-330/2020-1-SE. Mälardalen University, 2020.

Abstract

The problem of mission planning for multiple autonomous agents, including path planning and task scheduling, is often complex, especially when the number of agents grows or requirements include real-time constraints. In this paper, we propose a novel approach called MCRL that integrates model checking and reinforcement learning to overcome this difficulty. Our approach employs timed automata and timed computation tree logic to describe the autonomous agents' behavior and requirements, and trains the model by a reinforcement learning algorithm, namely Q-learning, to populate a table used to restrict the state space of the model. Our method provides a means to synthesize mission plans for autonomous systems whose complexity exceeds the scalability boundaries of exhaustive model checking, but also to analyze and verify synthesized mission plans to ensure given requirements. We evaluate the proposed method on various scenarios involving autonomous agents, as well as present comparisons with other methods and tools.

11.1 Introduction

Autonomous agents are systems that usually move and operate in a possibly unpredictable environment, can sense and act on it, over time, while pursuing their goals [1]. As this kind of systems bear the promise of facilitating people's daily lives and increasing safety and industrial productivity by automating repetitive tasks, autonomous technologies are drawing an increased attention from both researchers and practitioners. In an attempt to realize their functions, mission planning of autonomous agents, including path planning and task scheduling, is one of the most critical problems to solve [2]. As path-planning algorithms focus on calculating collision-free paths towards the destination, they do not handle requirements concerning logic and temporal constraints, e.g., delivering goods in a prioritized order, and within a certain time limit. In addition, when considering a group of agents that need to collaborate with each other and usually work alongside humans, the job of synthesizing correctness-guaranteed mission plans becomes crucial and more difficult.

In our previous work [3], we have proposed an approach based on Timed Automata (TA) and Timed Computation Tree Logic (TCTL) to formally capture the agents' behavior and requirements, respectively, and synthesize mission plans for autonomous agents by model checking. Our approach is successfully implemented in a tool called TAMAA, and shown to be applicable to solving the mission-planning problem of industrial autonomous agents. However, TAMAA alone is not scalable when the number of agents is large, as the state space of the model explodes when the number of agents grows.

The state-space-explosion problem is one of the most stringent issues when employing model checking [4] for verification, therefore many studies have explored ways of fighting it [5]. In this paper, we propose a novel method called **MCRL** that combines model checking with reinforcement learning [6] to restrict the state space in order to synthesize mission plans for large numbers of agents. Our method is based on UPPAAL [7] and leverages the model of autonomous agents generated by TAMAA. Instead of exhaustively exploring and storing states of the model, MCRL utilizes Monte Carlo simulations to obtain the execution traces leading to the desired states or deadlocks. Note that in TAMAA timing uncertainties are modeled by non-deterministic delays bounded from below as well as above, rather than by probability distributions. Therefore, the simulation is simply for randomly sampling execution traces. Then, a reinforcement learning algorithm, namely Q-learning [8], is employed to process the execution traces, and populate a Q-table containing the state-action pairs and their values. The Q-table is recognized as the mission plan

that we have aimed to synthesize, which is injected back into the old TAMAA model, forming a new model. As the Q-table regulates the behavior of the agent model, the state space is greatly reduced, which makes it possible to verify mission plans for large numbers of agents. Moreover, MCRL enables the model equipped with Q-tables to make best decisions when the task execution time and duration of movement are uncertain, which is not supported by TAMAA. As MCRL is based on formal modeling, it complements classic reinforcement learning algorithms with means to verify the synthesized mission plans against, for instance, safety requirements.

We select relevant scenarios involving autonomous agents in a construction site, and conduct experiments with MCRL, TAMAA, and UPPAAL STRATEGO [9] that is a tool often used for generating winning strategies for stochastic priced timed games. The experimental results show that MCRL performs better than TAMAA and UPPAAL STRATEGO, when the number of agents is greater than five. The time of synthesizing mission plans using MCRL increases linearly with the number of agents, whereas for the other two methods it increases exponentially.

To summarize, the contributions of this paper are:

- A novel approach called MCRL for synthesizing mission plans of large numbers of autonomous agents by reinforcement learning, combined with model checking the synthesis results.
- An evaluation of the scalability of MCRL via experiments conducted with tools such as TAMAA, UPPAAL STRATEGO, and MCRL, on relevant scenarios involving autonomous agents. The experimental results show that MCRL can scale to large numbers of agents that cannot be handled by other methods.

The remainder of the paper is organized as follows. In Section 11.2, we introduce the preliminaries of this paper. Section 11.3 describes the problem that we attempt to solve and its challenges, whereas in Section 11.4 we introduce our novel approach for taming the scalability of model checking, which combines the latter with reinforcement learning. In Section 11.5, we explain the experiments and their results on TAMAA, UPPAAL STRATEGO, and MCRL. In Section 11.6 we compare to related work, before concluding and outlining possible future work in Section 11.7.

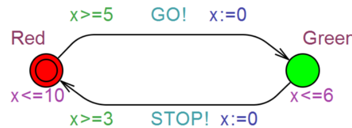


Figure 11.1: An example of a timed automaton of a traffic light

11.2 Preliminaries

In this section, we introduce timed automata, UPPAAL, UPPAAL STRATEGO, and reinforcement learning.

11.2.1 Timed Automata and UPPAAL

A *timed automaton* (TA) is a finite-state automaton extended with real-valued variables, called *clocks*, suitable for modeling real-time systems [10]. UPPAAL [7] is a tool for modeling, simulation, and model checking of real-time systems, which uses an extension of TA as the modeling formalism. Figure 11.1 depicts a simple example of a UPPAAL TA modeling traffic lights. Two locations `Red` and `Green` model the two colors of a traffic light. A clock variable `x` is used in the *invariants* (boolean expressions over clocks) on locations (e.g., x$<=$10) to enforce an upper bound of delaying in that location (in our case, after 10 time units, the automaton must leave location Red). Edges are directed lines used to connect locations, and they are decorated by guards, which are boolean conditions over clocks or discrete variables, which enable the automaton to traverse the respective edge once they evaluate to true. In our case, when x$>=$ 5, the TA may move from the Red to the Green location. In UPPAAL, there is a special type of location, namely *committed* (denoted by encircled c). It requires that time does not elapse in these types of locations and the next edge to be traversed needs to start from a committed location. Clocks can be reset over edges, e.g., x:= 0 in Figure 11.1, whereas discrete typed variables can be assigned values, accordingly, via updates on the edges, or via functions that are implemented by a subset of the C language in the declaration of the TA. A network of TA, $B_0 \parallel ... \parallel B_{n-1}$, is a parallel composition of n TA via *synchronization channels* (i.e., a! is synchronized with a? by handshake). In Figure 11.1, the edges are labeled with channels named `STOP` and `GO`, which synchronize this TA with other TA of vehicles.

The UPPAAL queries that we verify in this paper are properties of the form:

(i) **Invariance**: $A \Box p$ means that for all paths, for all states in each path, p is satisfied, (ii) **Liveness**: $A \Diamond p$ means that for all paths, p is satisfied by at least one state in each path, (iii) **Reachability**: $E \Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (iv) **Time-bounded Leads to**: $p \leadsto_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter; it is equivalent to the property: $A \Box (p \Rightarrow A \Diamond_{\leq t} q)$.

11.2.2 UPPAAL STRATEGO

UPPAAL has several branches that extends it to deal with various specific problems. UPPAAL STRATEGO [9] is a tool that integrates UPPAAL with two of its branches, that is, UPPAAL SMC [11] (statistical model checking) and UPPAAL TIGA [12] (policy synthesis for timed games). In this paper, we employ UPPAAL STRATEGO to solve the same mission-planning problem for autonomous agents, in order to compare the result with our MCRL approach. UPPAAL STRATEGO is designed to synthesize strategies for stochastic priced timed games. A game is a mathematical model of a system consisting of several players that compete in a common environment and aim to achieve their independent goals. Since it is based on UPPAAL, its modeling language is an extension of timed automata, which differentiates actions into two types: controllable and uncontrollable. The former ones are actions controlled by the players, whereas the latter ones are controlled by the environment. We refer readers to the literature [9] for details of this tool. A *strategy* is a policy of a player's actions for any possible situation that guides the player to reach its final goal. A *winning strategy* contains sequences of controllable actions that lead players to the states satisfying desired properties, regardless of the executed uncontrollable actions. In UPPAAL STRATEGO, one can synthesize winning strategies in form of: strategy S = control: P, where "=" is an assignment sign, P is the TCTL property to be met, and verify the synthesized strategies in the form of: P' under S, where P' is a stronger property that the model is verified against, with its behavior regulated by strategy S.

11.2.3 Reinforcement Learning

Reinforcement learning is a branch of machine learning aiming to calculate how agents should take actions in an environment, in order to maximize the accumulated reward obtained from the environment [6]. In this paper, we use one of the model-free reinforcement learning algorithms called *Q-learning* [8],

which is usually adopted to learn policies that indicate agents the actions to take at different states. A policy is associated with a state action value function called *Q function*, where “Q” stands for “quality”. The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \max_{a'} q^*(s',a')], \tag{11.1}$$

where $q^*(s,a)$ represents the expected reward of executing action a at state s, $\mathbb{E}$ denotes the expected value function, $R(s,a)$ is the reward obtained by taking the action a at state s, γ is a discounting value, s' is the new state coming from state s by taking action a, $\max_{a'} q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair (s',a'). The equation means that the expected reward of the state-action pair (s,a) is the sum of the current reward and the discounted maximum future reward. As the learning process iterates, the Q-value of each state-action pair converges to the maximum Q-value, i.e., q^*, and the parameters are updated using gradient descent [13]. Although Q-learning is a model-free algorithm, the learning process often relies on a simulation environment that depends on the form of the model. In this paper, we use the simulation function in UPPAAL to gather the information of state-action pairs, and invoke the Q-learning algorithm to populate a Q-table that stores state-action pairs and their Q-values.

11.3 Problem Description

In this section, we introduce an industrial use case of an autonomous quarry, containing various autonomous vehicles, e.g., trucks, wheel loaders, etc. For example, as shown in Figure 11.2, we consider the mission of transporting stones in a quarry site, where a wheel loader digs and loads stones, and trucks transport stones. They need to carry the stones from stone piles to the primary crushers, where stones are crushed into fractions, and proceed to carry the crushed stones to the secondary crushers, which is the destination. During this process, the vehicles must avoid static obstacles (e.g, holes and rocks on the ground, larger than given sizes) and go to the charging point when their battery level is low. In an autonomous quarry, all the operations are performed automatically without human intervention, and the vehicles are autonomous agents that we call *agents* for short, in this paper. To achieve their goal, respectively, the agents need to be able to calculate collision-free paths and schedule their tasks efficiently. Hence, our research problem involves task scheduling,

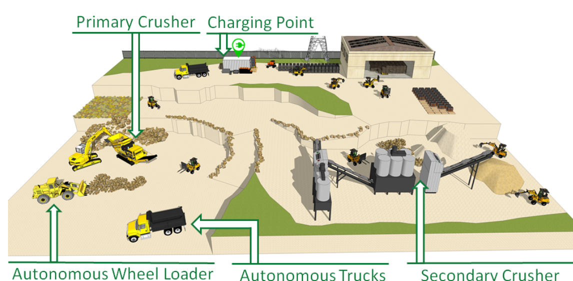


Figure 11.2: An example of an autonomous quarry

path planning and following, and collision avoidance for multiple autonomous agents. In our previous work [14], we have proposed a two-layer framework for the design of formal models of agents, where task scheduling and path planning belong to the so-called static layer, whereas the path following and avoidance of dynamic obstacles, including the case of overlapping paths of multiple agents, is being dealt with in the dynamic layer. In this paper, we assume that the collision avoidance of dynamic obstacles functions correctly, and focus on the static layer for synthesizing verifiable mission plans.

11.3.1 Problem Analysis

For simplicity, henceforth, we call the problem of path planning and task scheduling for agents as **mission planning**. Path planning deals with computing collision-free paths that visit all required target positions (a.k.a. milestones), via efficient algorithms such as Theta* [15] and RRT [16]. We adopt the Theta* algorithm in this paper, since the environment in the problem is a 2-D map, and the algorithm is especially good at generating smooth paths with any-angle turning points in 2-D maps. After the paths are calculated, the agents need to know the assignment and execution order of tasks. For instance, digging stones must be carried out at stone piles before the stones are unloaded into the primary crushers. In this case, digging stones and unloading stones are two tasks, and their execution positions and order must be correct. Additionally, as the machines must guarantee a certain level of productivity, the work has to be completed within some given time. As our solution aims to be general, regardless of the exact type of agents, we formulate the requirements generically, and categorize them as follows:

- *Milestone Matching*. Tasks must be performed at the right milestones.
- *Task Sequencing*. The task execution order must be correct.
- *Timing*. Tasks must be done within prescribed times.
- *Event Reaction*. Some special tasks are only triggered by events under certain circumstances, e.g., when the battery level is low, the agents must go to charge themselves.

The task-scheduling problem in this paper is similar to a classic scheduling problem called *job-shop* problem [17], which consists of a finite set of jobs to be processed on a finite set of machines. Each job is a sequence of tasks to be executed in a certain order and no tasks can be preempted once started. Each machine can process at most one task at a time and the execution time varies for tasks, but it is fixed. The objective is to assign jobs to machines and decide their starting times in order to minimize the total execution time of all jobs. The problem is NP-hard, so even a simple instance with very restrictive constraints remains difficult to solve [18]. Although the task scheduling in this paper shares many similarities with the job-shop problem, e.g., tasks are non preemptive, our problem has some unique challenges that we introduce in the following section.

11.3.2 Uncertainties and Scalability of Mission Planning

The classic job-shop problem is deterministic as the information is known and fixed. However, the task-scheduling problem in this paper contains two types of uncertainties, i.e., the uncertain execution time of tasks and uncertain duration of agent movement.

- *Uncertain execution time of tasks*. The execution time of tasks is modeled by time intervals between the BCET (best-case execution time) and WCET (worst-case execution time), which are usually different.
- *Uncertain movement time*. The traveling time between milestones of any agent is not fixed, due to the fact that the destination milestone can be occupied at some time, and thus the agent that is approaching it has then to wait until the destination is available, and the waiting time is uncertain.

These features make our problem more difficult than the classic job-shop problem. When the number of agents increases, the complexity of the problem grows exponentially.

Table 11.1: Evaluation of TAMAA and UPPAAL STRATEGO

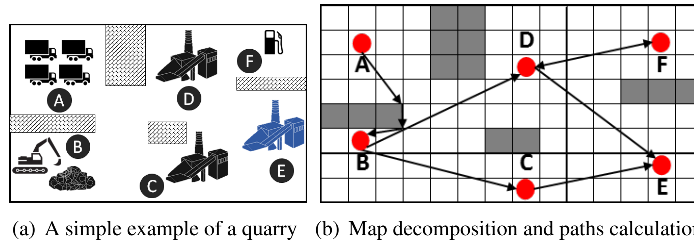
	Number of Agents	Number of Explored States	Time
TAMAA	4	2,058,132	20160 s
	5	Out of Memory	Out of Memory
UPPAAL STRATEGO	2	12,031	2670 s
	3	Out of Memory	Out of Memory

In our previous work [3], we propose a timed-automata-based approach called TAMAA to solve this problem. Although the approach manages to generate mission plans satisfying complex requirements, when the number of agents increases to 5, model checking the TAMAA model exhausts the existing memory due to the state-space explosion problem of model checking [3, 4]. To compare with a similar existing approach, we also employ UPPAAL STRATEGO [9] to synthesize mission plans by verifying the model of TAMAA in this tool (Section 5). Researchers have utilized UPPAAL STRATEGO to solve similar scheduling problems like ours, for e.g., cruise control [19], and floor heating [20], which involve assigning motions to "players" in the environment, to obtain winning strategies. Thus, UPPAAL STRATEGO is considered to be suitable to solve such task-scheduling problems. However, UPPAAL STRATEGO is only able to generate results when the number of agents is less than 3, as it is shown in Table 11.1. In a nutshell, task scheduling for multiple autonomous agents, as an NP-hard problem, remains unsolved when the number of agents is large.

11.4 MCRL: Combining Model Checking and Reinforcement Learning in UPPAAL

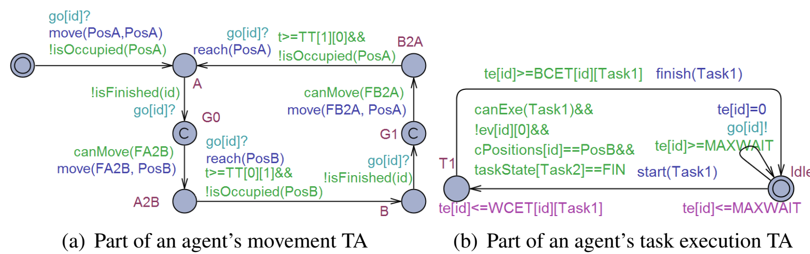
In this section, we introduce our novel approach called MCRL for mission planning of multiple autonomous agents, which combines model checking with reinforcement learning to alleviate the state-space-explosion problem. The TA model in MCRL originates from TAMAA, therefore, we first briefly introduce TAMAA in the following section to lay the foundation of this method. The formal definitions of the movement and task execution in TAMAA, as well as the model generation algorithms are described in our previous work [3], which the interested reader is referred to for details[1].

[1] A demo of TAMAA is in https://doi.org/10.5281/zenodo.3614128



(a) A simple example of a quarry (b) Map decomposition and paths calculation

Figure 11.3: An example of an autonomous quarry



(a) Part of an agent's movement TA (b) Part of an agent's task execution TA

Figure 11.4: The TA model of the example in Figure 11.3

11.4.1 Timed-Automata-Based Model for Mission Plan Synthesis

We elaborate the TA model in TAMAA by an example illustrated in Figure 11.3(a). In an autonomous quarry, there are four autonomous trucks starting from milestone A, aiming to transport stones at milestone B, to the primary crusher at milestone C or D, and eventually go to the secondary crusher at milestone E. There are also autonomous wheel loaders working at milestone B, digging stones and loading them into the trucks. A charging point is located at milestone F, where all the vehicles go for charging when their battery-level is low.

Initially, the environment is decomposed into a Cartesian grid and the Theta* algorithm [15] is executed to calculate the shortest paths among milestones A - F (See Figure 11.3(b)). Note that the trucks only need to choose one primary crusher at position C or D, to unload stones. Next, a TA-model is automatically generated by TAMAA, based on the decomposed environment. For brevity, an example of the TA model is only partly shown in Figure 11.4(a). It models

the movement of autonomous trucks between milestones A and B. The initial location of the automaton has only one outgoing edge to location A, indicating that milestone A is where the truck starts. Locations A2B and B2A are created to count the duration of traveling between A and B. Variable TT[m_1][m_2] is the travelling time between milestones m_1 and m_2. Locations G0 and G1 are committed locations that do not cost any time and are used to diverge the movement to multiple targets. Since some of the milestones are not accessible when they are occupied, the guard function "isOccupied" is utilized (see Figure 11.4(a)) to judge if the milestones are occupied or not. When the function returns false, the edge is enabled but does not trigger the transition, which means that the agent can stay at this location rather than go to the target. Therefore, the incoming edges of locations A and B are labelled with channels "go[id]?", where "id" represents the index of the agent, and it synchronizes the movement TA with the tasks execution TA.

When an agent is at a milestone, it has three options for the next motion: staying, moving, or executing tasks. TAMAA generates a TA for tasks execution that models these behaviors. This TA is partly depicted in Figure 11.4(b), with location Idle representing the no-operation task, where the agent is allowed to move. The invariant and self loop of location Idle represent the time unit of scheduling a moving action. Every "MAXWAIT" time unit, the tasks execution TA informs the movement TA that the agent is ready to move. Location T1 represents the task "loading", and the guard on its incoming edge regulates that it must be carried out at milestone B and after task 2, finished provided that the charging event does not occur. Location T1 has an invariant that indicates that the actual execution time of task "loading" must not exceed its WCET. Similarly, the guard on the outgoing edge of T1 ensures that the agent leaves the location when the execution time is no longer less than BCET.

After the resulting TA model is verified in UPPAAL, execution traces indicating the order of visiting milestones and operating tasks are generated. Since UPPAAL provides three types of execution traces, i.e., the shortest, the fastest, and random ones, we can generate mission plans that take the least number of steps (shortest), or the shortest time (fastest), or random. However, the verification is based on exhaustive model checking, which means that the entire state space is built and stored during the process. Therefore, the number of states of the model grows exponentially as the number of agents increases, and thus the computation time and memory consumption increase dramatically, as it is shown in Table 11.1. In the following, we show how we alleviate this shortcoming, by applying a reinforcement learning algorithm to reduce the state space of model checking the TA model.

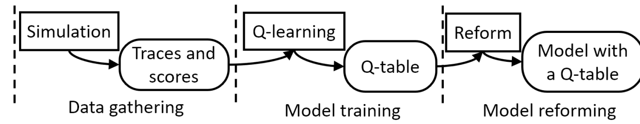


Figure 11.5: The process of creating a model using a Q-table

11.4.2 MCRL Method Description

In order to alleviate the state-space explosion problem, MCRL adopts random simulation instead of exhaustive model checking, and trains the model by the Q-learning algorithm. Figure 11.5 depicts the process of the method. First, in the data-gathering phase, we obtain the execution traces of the model by Monte Carlo simulation in UPPAAL. We assign rewards to the state-action pairs of the execution traces that satisfy the desired properties, and penalties to the ones containing deadlocks. The traces that either hold the properties or contain deadlocks are ignored and not used in the next phases. Thereafter, in the model-training phase, we adopt the Q-learning algorithm, which is implemented as Java program, to process the traces and populate a Q-table, which is then used to form a new model whose state space is restricted. Details of this approach are presented in the following.

Model Design and Data Gathering

To differentiate between the state of TA and the state of Q-tables, we define Q-state and Q-action as follows:

Definition 1 (Q-state). *A Q-state is defined as a tuple:*

$$\mathcal{QS} =< TP, MATCH >,$$

where TP is a real number denoting the time of leaving this state, $MATCH$ is a tuple $< RT, CT, CP, EV, ST >$, where

- *RT is an integer denoting the number of rounds for finishing all tasks,*
- *CT is an integer denoting the index of the current task,*
- *CP is an integer denoting the index of the current milestone,*
- *EV is a set of Boolean values of events, occurred or not,*

- *ST is a set of integers of EST (execution status of tasks) of all the agents in the environment.* □

Definition 2 (Q-action). *A Q-action is defined as a tuple:*

$$\mathcal{QA} =< BT, WT, MT, TT >,$$

where,

- *BT is a real number denoting the BCET of the action,*
- *WT is a real number denoting the WCET of the action,*
- *MT is an integer denoting the type of the motion,*
- *TT is an integer denoting the target of the motion.* □

"TP" in Definition 1 is created to distinguish "meaningless" execution traces of agents that simply move around and consume plenty of time but do not complete tasks. The Q-states that have the same values of other attributes but own a much larger value of "TP" can be omitted. Note that "ST" in Definition 1 represents the execution status of tasks (EST) of all agents in the environment. It has three possible integer values, i.e., 0: unfinished, 1: finished, or 2: will be finished by the time the current agent arrives at the milestones where other agents locate. As each agent owns a Q-table, when they need to make a decision, i.e., which milestone to go, or which task to execute, they must be aware of the EST of other agents to avoid unnecessary waiting. "MT" in Definition 2 has two possible values, i.e., 0: movement, 1: execution. Correspondingly, "TT" can be the index of the target milestone, or the index of the next task.

All the attributes of a Q-state and a Q-action can be elicited from the TA model generated by TAMAA, and thus, we create a 2-dimensional array in the global declaration of the TA model in UPPAAL to represent the Q-table for each of the agent mode. The state-action pairs in the Q-tables are calculated and stored during the random simulation of the model. UPPAAL 4.1.22[2] provides a new function of simulation that prints information only when certain predicates are true. For example, in the following query, the model is simulated 1000 rounds and 100 time units for each round. Only when the predicate following

[2] UPPAAL 4.1.22 was published in March 2019 on http://www.uppaal.org/

the simulation query is true, i.e., the Boolean variable "taskAllFinish" turns true, the information within the curly parentheses ($\{\ldots\}$) is printed.

$$\texttt{simulate}[<= 100; 1000]\{...\} : \texttt{taskAllFinish} == \texttt{true}$$

By using this function, we can control the simulation to print data when all tasks are finished (good traces), or any of the agents is stuck in a deadlock (bad traces). At the end of each round of the simulation, if the predicate is satisfied, rewards (positive values) are assigned to the state-action pairs in the trace by the functions in the TA model; if a deadlock occurs, penalties (negative values) are assigned to them in a similar way. More precisely, the reward has a value of $MAX - CTime$, where MAX is the maximum simulation time, $CTime$ is the time point of finishing all tasks, whereas penalties have the same fixed value. In this way, the traces that reach the states that satisfy the predicates faster would get higher rewards and thus are enhanced by Q-learning.

There are several things about the simulation that deserve further explanation. In UPPAAL, the simulation query subsumes Monte Carlo simulation to simulate the model, which is originally designed for statistical model checking [11]. However, in this paper, we do not adopt this feature of UPPAAL but only utilize the Monte Carlo simulation to explore the state space of the model, and the only two uncertainties in the problem, e.g., uncertain task execution and movement times, are modeled as time-bounded delays that follow a uniform probability distribution. One can change it to an arbitrary choice of time-bounded delays or other probability distributions and still use MCRL to solve the problem. Additionally, the simulation time of each round should not be shorter than the shortest time needed to finish tasks, otherwise the predicate remains false and thus no good trace can be gathered in the simulation. The number of simulation rounds should be set properly so that the gathered data is not only enough for training the model, but also not too large, which would entail unnecessarily long time to process it. When the simulation finishes, UPPAAL prints the state-action pairs into a file, which is used in the model-training phase.

Model Training and Reforming

After the state-action pairs are formed in the simulation, we input those data into the Q-learning algorithm, which is implemented as Java program, to populate a Q-table. We illustrate the format of the Q-table as follows:

$$|\text{Q-state}|\text{Q-action}|\text{Q-value}|$$

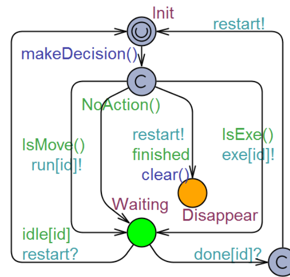
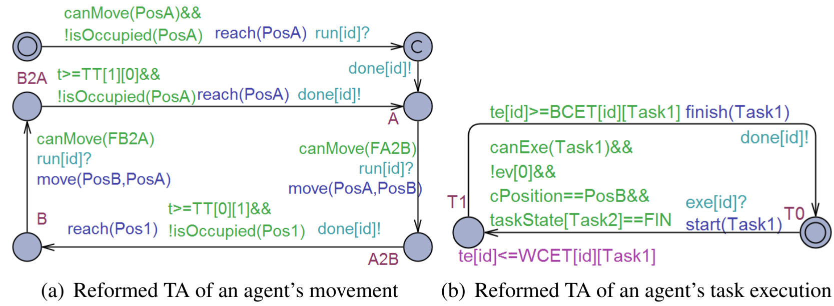


Figure 11.6: The conductor TA in the new model with a Q-table

As aforementioned, the Q-tables for agents are stored in a two-dimensional array of the TA model. After running the Q-learning algorithms, Equation 11.1 guarantees that the Q-values of the state-action pairs are accumulated and converged.

In the model-reforming phase, a new TA model, which we call conductor, is designed for each of the autonomous agents, which looks up the agent's Q-table and sends controlling commands. Since there is no centralized control in the environment, each agent model is equipped with one conductor. However, the conductor contains the Q-tables of all agents in order to decide which one has the priority to act, when multiple agents intend to perform some concurrent actions. Figure 11.6 depicts the TA model of conductors. The initial location `Init` is urgent to ensure that whenever the agent is ready, it is scheduled immediately. The function `makeDecision()` looks up the Q-table and chooses the state-action pair that owns the highest value among those that match the current state of the agent. Note that, here we only need to compare the attributes in "$MATCH$" but not "TP", because the former is enough to represent the states of the agent and environment. If the chosen action is "execution", the conductor sends an "executing" command to the task execution TA via channel "exe[id]". If the chosen action is "movement", the conductor looks up other agents' Q-tables to obtain their intentions of actions. If they also intend to go to the same milestone where agents are mutually exclusive, the one with the highest value of state-action pair is allowed to move, whereas others have to wait until the former finishes scheduling. Whatever the command is, the conductor TA transfers to the location `Waiting` to wait until the agent finishes its action and responds via the synchronization channel "done[id]".

The fact that locations, expect locations "`Disappear`" and "`Waiting`",



(a) Reformed TA of an agent's movement (b) Reformed TA of an agent's task execution

Figure 11.7: Reformed TA model

are either urgent or committed guarantees that all agents are scheduled simultaneously. Meanwhile, UPPAAL sets the order of running the conductors to be arbitrary, which means agents could act in any order. However, the formal verification of the model equipped with Q-tables can prove that no matter what the acting order is, agents are guaranteed to satisfy the desired properties. This is what traditional RL algorithms cannot provide.

Consequently, the original TA of movement and task execution (see Figures 11.4(a) and 11.4(b) as examples) need to be slightly adjusted. As depicted in Figures 11.7(a) and 11.7(b), the edges with functions `move()` and `start()` are labelled with channels "run[id]?" and "exe[id]?", respectively. In those two functions, a Boolean variable "idle[id]" is turned to *false*, indicating that the agent is scheduled to start working. However, if the target position is occupied at the moment and multiple agents are not allowed at this milestone, the movement TA should not transfer. Hence, the channel "run[id]" is broadcast so that it does not block the transition in the conductor TA, and the variable "idle[id]" remains *true* because the function `move()` is not invoked.

In this case, the conductor TA needs to be informed when the position is released in order to re-schedule the agent. When the action finishes, the conductor leaves location `Waiting` and moves back to the initial location to start another round of scheduling. As the times of such actions are not determined, the conductor does not know when to restart. Hence, the edges with functions `finish()` and `reach()` in the task execution TA and movement TA are synchronized with the conductor TA via channel "done[id]", so that whenever an agent completes an action, its conductor restarts. The conductor TA could also go back to its initial location via the edge labelled with a broadcast channel "restart?" and a guard "idle[id]", indicating that some other agent has changed

its state, and if the current one is idle, it can be re-scheduled. A Boolean variable "finished" is used in the conductor TA. When the agent finishes the requested rounds of work, this variable turns to *true* on the edge going to the location `Disappear`, and the milestone occupied by this agent is released, indicating that it has left the site and stopped. This edge is also labelled with the channel "restart!" to inform other agents for re-scheduling.

Mission Plan Synthesis and Analysis

By introducing the conductor TA, the behavior of the autonomous agents is restricted by the Q-table. Hence, if the Q-table is formed by using the state-action pairs satisfying certain predicates, the reformed model is supposed to satisfy the predicates. For example, in the data-gathering phase, the simulation query is designed as follows:

$$\texttt{simulate[<=T; R] \{...\}: forall(i:int[0,N-1]) work[i]} \geq \texttt{X},$$

where T is the simulation time of each round, R is the number of simulation rounds, N is the number of agents, X is the requested rounds of work. In the case of autonomous trucks, one round of work means starting from the stone pile and eventually unloading stones at the secondary crusher as it is shown in Figure 11.2. The predicate regulates that if the N agents accomplish X rounds of work, the information in the parenthesis ({...}), i.e., the state-action pairs and their rewards/penalties, is printed. Hence, when the TA model is verified in UPPAAL, properties of the following form:

$$\texttt{A}\Diamond\,\texttt{forall(i:int[0,N-1]) work[i]} \geq \texttt{X}, \tag{11.2}$$

should be satisfied, which we demonstrate in Section 11.5. Meeting this kind of properties proves that the Q-table serves as the mission plan that we intend to synthesize, and guides the agents to accomplish a requested amount of work. Additionally, one can also verify properties of the following form:

$$\texttt{A}\Box\,\texttt{forall(i:int[0, M-1]) positionOccupied[i]} \leq \texttt{1} \tag{11.3}$$

$$\texttt{batter==low} \;--> \texttt{movement.charging} \;\&\&\; \texttt{x} \leq \texttt{L} \tag{11.4}$$

Equation 11.3 requires that milestones are never occupied by multiple agents. Equation 11.4 requires that the agent goes to the charging point within L time units, when its battery level is low. One can design their own properties, or TA model, to express and verify specific requirements. These properties are impossible to be verified by traditional model checking alone in the cases containing large numbers of agents, due to the exponentially grown state space.

Table 11.2: Tasks for the autonomous agents in the experiment

	Task	BCET	WCET	precondition
Truck	Load	1	4	none
	Unload	4	4	Load
	Charge	15	15	none
Wheel loader	Dig	2	2	none
	Unload	1	4	Dig
	Charge	15	15	none

11.5 Experimental Evaluation

In this section, we evaluate our approach by conducting experiments on MCRL, TAMAA, and UPPAAL STRATEGO to make a comparison. The experiments are conducted in UPPAAL 4.1.22 and UPPAAL STRATEGO 4.1.20-7, on a laptop running an Intel Core i5 processor with 16 GB of RAM and a 64-bit Windows OS. The environment model in this experiment is the one depicted in Figure 11.3(a), containing 4 static obstacles, 6 milestones, and several autonomous trucks and 1 autonomous wheel loader. To make a comparison with TAMAA and UPPAAL STRATEGO, we vary the number of agents from 2 to 6. The tasks and their execution times for autonomous trucks and wheel loader are shown in Table 11.2.

Experimentation using TAMAA. After configuring the agents, tasks, and environment in the TAMAA tool, we obtain the TA model of task execution, movement, and monitor for the battery-low event. To synthesize the mission plan that transfers all the stones to the secondary crusher with the minimum time consumption, we verify the model in UPPAAL and select the fastest diagnostic trace. The TCTL query designed for the verification is as follows:

$$E\Diamond\ (\texttt{stone==0}\ \&\&\ \texttt{time}\leq\texttt{LIMIT}), \tag{11.5}$$

where the variable “stone” represents the volume of the stone pile, whose value is updated in the function “finish()” in the task execution TA, and “time ≤ LIMIT” regulates the time limit of finishing the job. The verification results[3] show that TAMAA can generate mission plans that guide the agents avoid static obstacles and carry all the stones to the destination. However, this approach can only synthesize a certain type of mission plans, e.g., fastest, shortest, or random, as UPPAAL provides these three types of diagnostic traces. When the

[3]Graphic mission plans in TAMAA: http://doi.org/10.5281/zenodo.3731960

execution times of tasks are uncertain, these types of mission plans are not sufficient to handle all situations.

Experimentation using UPPAAL STRATEGO. In order to synthesize mission plans in UPPAAL STRATEGO, the TA model in TAMAA needs to be adjusted slightly. See Figure 11.4(a) as an example, where edges from location A2B to location B and from location B2A to A in the movement TA are changed to "uncontrollable" ones, as they are controlled by the environment. Similarly, in the task execution TA, the incoming edges of location Idle are changed to "uncontrollable". Thereafter, we verify the model against queries as follows:

$$\texttt{strategy MP = control: A}\Diamond\ \texttt{stone==0} \tag{11.6}$$

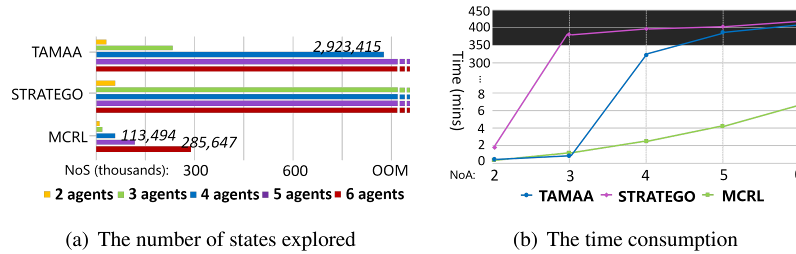
$$\texttt{E}\Diamond\ \texttt{(stone==0}\ \&\&\ \texttt{x}\leq\texttt{MAXTIME) under MP} \tag{11.7}$$

Query 11.6 utilizes a special syntactical keyword of UPPAAL STRATEGO "control" to synthesize strategies that enable the model to transfer all the stones to the secondary crusher under any circumstances (i.e., A$\Diamond$). Query 11.7 verifies the model to see whether the agents are able to transfer stones within a time limit (i.e., "x$\leq$MAXTIME) under MP"), when their behaviors are restricted by the strategy (i.e., "under MP"). These queries provide a means of synthesizing and optimizing mission plans that handle the uncertain times of task execution and movement, which is better than TAMAA. However, as UPPAAL STRATEGO still adopts exhaustive model checking to generate mission plans (strategies) by queries like Query 11.6, the state-space explosion problem is inevitable when the system is large and complex.

Experimentation using MCRL. In this experiment, we train and reform the TA model of TAMAA in the way described in Section 11.4.2. Then, we synthesize mission plans for 2 to 6 autonomous agents. Figure 11.8(a) shows the comparison of the number of explored states in the verification using different methods, where "OOM" means the verification runs out of memory and fails to generate a result. As shown in Figure 11.8(a), MCRL is able to generate a result for all the cases and explores much less states than the other two methods. This demonstrates that the new approach is applicable and scalable to solve the mission-planning problem for larger numbers of agents. We experiment up to 6 agents, however we believe that MCRL is able to handle even larger numbers of agents.

11.5.1 Discussion

From the experimental results we can conclude that MCRL can generate results for up to 6 agents, TAMAA for maximum 4 agents, and UPPAAL STRATEGO



(a) The number of states explored

(b) The time consumption

Figure 11.8: Experimental result of the algorithm performance of synthesizing mission plans for different numbers of agents using three methods

for maximum 2 agents (see Figure 11.8(a)). Figure 11.8(b) shows the computation time of synthesizing mission plans using different methods. Since the difference between times are significantly large, in order to show the data in one graph, the Y-axis is not entirely equidistant, as from 8 we skip numbers. Since TAMAA and UPPAAL STRATEGO fail to generate results when agents are more than 4 and 2 respectively, the black portion of the graph indicates that the methods exhaust memory and return an "out of memory" error after large amounts of time, respectively. The computation time of MCRL is the sum of computing all phases, including data gathering, model training and reforming. As the number of agents grows, the time increase of computation is nearly linear. In the case of 3 agents, TAMAA costs the least time, as UPPAAL STRATEGO and MCRL consider all the situations of uncertain task execution and movement times, which are not dealt with by TAMAA. In the case of 4 agents, TAMAA can still generate results but costs more than 5 hours, whereas MCRL only needs nearly 3 minutes.

Beside the ability of handling larger number of agents, MCRL also provides a way to analyze the synthesized mission plans. Given the model with a Q-table, we can inspect sample mission plans via simulation query as follows:

```
simulate[<=45; 2]{ position, task+6},
```
(11.8)

where tasks and positions are encoded as different levels, and the simulation runs 2 rounds and 45 time units for each round. The result of the simulation query is depicted in Figure 11.9, which indicates that the agent probably goes to the primary crusher at milestone `C` or `D` (see Figure 11.3(a)), to carry out the unloading task. It is due to the fact that, in case either milestone is being occupied, the agent knows to go to the other one to avoid unnecessary waiting.

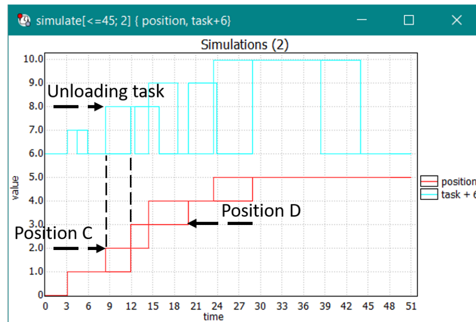


Figure 11.9: Two samples of mission plans

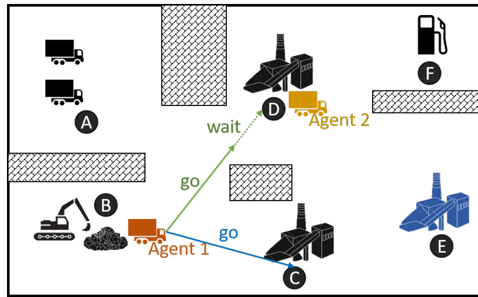


Figure 11.10: A scenario where agent 1 learns in the training phase

This “intellegence” is obtained through the model-training phase, which is one of the benefits of adopting Q-learning. One can design various queries to analyze the synthesized mission plans in this integrated method, which is another contribution of MCRL.

By verifying Query 11.9, we can get the counter-example of the query that enables one to understand how the choice is made.

$$\texttt{A}\square\ \texttt{agent[1].unload==FIN}\ \textit{imply}\ \texttt{movement1.C} \tag{11.9}$$

As illustrated in Figure 11.10, when agent 1 finishes the loading task, agent 2 is occupying the primary crusher at position D and unloading stones. At this moment, if agent 1 goes to position D, it needs to predict whether agent 2 is

still there, which entails that agent 1 has to wait. To achieve this, we employ the attribute "ST" (execution status of tasks) in Definition 1.

The moment agent 1 finishes its task at position `B`, it sends a request to obtain the execution status of the agent working at position `D`, which contains two elements: execution status (ES) and worst-case-execution-time of the current task ($WCET$). Based on the movement TA of agent 1, it is aware of its traveling time of reaching position `D`. Hence, ES of agent 2 can be easily predicted by the following formula:

$$ES_2(c + \mu_1) = \begin{cases} FIN, & ES_2(c) == FIN, \\ UFIN, & ES_2(c) \neq FIN \;\&\; c + \mu_1 < WCET_2, \\ WFIN, & ES_2(c) \neq FIN \;\&\; c + \mu_1 \geq WCET_2, \end{cases} \tag{11.10}$$

where c is the current time, and μ_1 is agent 1's traveling time to position `D`. Formula 11.10 means: (i) if agent 2's current task has finished at the moment, after the traveling time of agent 1, it is still "finished", or, (ii) if the future time point $(c + \mu_1)$ is less than the $WCET$ of agent 2's current task, it is "unfinished", otherwise (iii) it "will-be-finished". This formula provides a conservative prediction if the $WCET$ is different from the $BCET$ of the task. One can change $WCET$ in Formula 11.10 with $BCET$ to make aggressive predictions.

Once the states of the model are distinguished in this way, the learning algorithm is able to gradually acquire the optimal decisions for different situations, after multi-rounds of simulation. For example, in the data-gathering phase, we obtain the state-action pairs of agents going to positions `C` and `D`. The learning algorithm assigns higher values to the ones with less time consumption, therefore, like the situation in Figure 11.10, when the predicted execution status of agent 2 is unfinished, agent 1 going to position `C` is "reinforced" because it is faster. Moreover, Query 11.11, a modified version of Query 11.9, can be satisfied, which means the observation in the sample is generally held by the mission plan.

$$\begin{aligned} &(\texttt{agent[1].load} == \texttt{FIN} \;\&\&\; \texttt{agent[2].unload} == \texttt{UFIN}) \\ &\quad -- >(\texttt{agent[1].unload} == \texttt{FIN}\; \textit{imply}\; \texttt{movement1.C}) \end{aligned} \tag{11.11}$$

Besides this example, one can specify various requirements by using CTL/TCL queries, and apply MCRL to synthesize mission plans and verify them by model checking. To the best of our knowledge, the ability of synthesizing verifiable mission plans for large numbers of agents is not provided by any existing solution in the literature.

Although promising, one observation of MCRL is that if the simulation rounds in the data gathering phase are not enough, and thus do not obtain

enough data, the method is unable to synthesize valid mission plans, even when there exists one solution in the original model. Currently, the number of simulation rounds is decided based on the experience of designers, and a method to infer the number is needed in the future work. However, according to the experiments (see Figure 11.8(b)), we know that, even including all phases of MCRL, the total time consumption is much less than other two methods when the number of agents grows.

11.6 Related Work

Recently, there has been a rising interest in policy synthesis for autonomous systems. Wang et al. [21] propose a novel POMDP (Partially Observable Markov Decision Processes) formulation to synthesis policies over a vast space of probability distributions so that their approach is capable of handling uncertain obstacles. Bouton et al. [22] also employ POMDP for modeling, and their solution enables the autonomous vehicles to adapt to the behavior of other agents. Nikou et al. [23] propose an automata-based solution for controller synthesis of multi-agent path planning, where Metric Interval Temporal Logic (MITL) is used to describe each agent's individual high-level specification. In contrast to these studies, our approach combines model checking and reinforcement learning so that both merits benefit our solution that proves to be accurate and scalable.

The combination of formal methods and learning algorithms is a recent trend that attracts a large body of research work. Li et al. [24] utilize the expressiveness of formal specification languages to capture complex requirements of robotic systems to construct reward functions of reinforcement learning so that they are interpretable. Bouton et al. [25] propose a generic approach to enforce probabilistic guarantees on agents trained by reinforcement learning. Mason et al. [26] present an assured reinforcement learning algorithm using abstract Markov decision processes and probabilistic model checking to establish abstract policies for autonomous agents that are formally verified. As aforementioned, UPPAAL STRATEGO is a new branch of UPPAAL designed by David et al. [9], which adopts reinforcement learning algorithms to refine the synthesized strategies for winning priced timed games. However, as different from these studies, our approach focuses on using reinforcement learning to replace exhaustive model checking for mission-plan synthesis of multi-agents, so that the state-space explosion is alleviated.

To the best of our knowledge, the first attempt to solve the state-space-

explosion problem of model checking using reinforcement learning is done by Behjati et al. [27]. These authors propose a bounded rational verification approach for on-the-fly model checking. However, this method is limited to non-timing LTL properties.

11.7 Conclusion and Future Work

We present a novel mission-plan synthesis method called MCRL that can handle large numbers of autonomous agents. The method adopts formal modeling to capture the behavior of autonomous agents and Q-learning to train the model and synthesize mission plans in the form of Q-tables. We demonstrate MCRL's ability of handling multiple agents by an experiment, and compare the result with TAMAA and UPPAAL STRATEGO. The experimental results show that the computation time of MCRL increases linearly with the number of agents, whereas the other two approaches show an exponential increase of their computation time, respectively. MCRL is also able to cope with uncertain task execution and movement times, which is not supported by exhaustive model checking in TAMAA. We present means for verifying and analyzing the synthesized mission plans using model checking to ensure safety-critical requirements. As the current approach does not consider unforeseen situations such as undetected obstacles, one direction of the future work is to introduce statistical model checking into our method to cope with probabilistic situations. Another possible direction will focus on integrating Q-learning directly into the generation of the state space with UPPAAL, and possibly on applying other machine learning or AI algorithms to tame verification scalability or guide the model checking itself.

Acknowledgment

The research leading to the presented results has been undertaken within the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, funded by the Swedish Knowledge Foundation, grant number: 20150022.

Bibliography

[1] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 21–35. Springer, 1996.

[2] PR Chandler and Meir Pachter. Research issues in autonomous control of tactical uavs. In *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*. IEEE, 1998.

[3] Rong Gu, Eduard Paul Enoiu, and Cristina Seceleanu. Tamaa: Uppaal-based mission planning for autonomous agents. In *The 35th ACM/SIGAPP Symposium On Applied Computing SAC2020, 30 Mar 2020, Brno, Czech Republic*, 2019.

[4] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School*, pages 1–30. Springer, 2011.

[5] Radek Pelánek. Fighting state space explosion: Review and evaluation. In *FMICS Workshop*. Springer, 2008.

[6] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 2. MIT press Cambridge, 1998.

[7] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, 3098:87–124, 2004.

[8] Christopher J.C. Hellaby Watkins. Learning from delayed rewards. 1989.

[9] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal stratego. In *TACAS*. Springer, 2015.

[10] R. Alur and D. Dill. Automata for Modeling Real-time Systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990.

[11] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. *arXiv preprint arXiv:1208.3856*, 2012.

[12] Gerd Behrmann, Alexandre David, Emmanuel Fleury, Kim Larsen, Didier Lime, and Ecole Nantes. Uppaal-tiga: Time for playing games! (tool paper). 2007.

[13] Mykel J Kochenderfer. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[14] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Towards a two-layer framework for verifying autonomous vehicles. In *NASA Formal Methods Symposium*, pages 186–203. Springer, 2019.

[15] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Artificial Intelligence Research*, 39, 2010.

[16] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[17] Henry Fisher. Probabilistic learning combinations of local job-shop scheduling rules. *Industrial scheduling*, pages 225–251, 1963.

[18] Yasmina Abdeddaı, Eugene Asarin, Oded Maler, et al. Scheduling with timed automata. *Theoretical Computer Science*, 354(2), 2006.

[19] Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Safe and optimal adaptive cruise control. In *Correct System Design*. Springer, 2015.

[20] Kim G Larsen, Marius Mikučionis, Marco Muniz, Jiří Srba, and Jakob Haahr Taankvist. Online and compositional learning of controllers with application to floor heating. In *TACAS*. Springer, 2016.

[21] Yue Wang, Swarat Chaudhuri, and Lydia E Kavraki. Bounded policy synthesis for pomdps with safe-reachability objectives. In *International Conference on Autonomous Agents and Multi Agent Systems*. IFAAMS, 2018.

[22] Maxime Bouton, Akansel Cosgun, and Mykel J Kochenderfer. Belief state planning for autonomously navigating urban intersections. In *Intelligent Vehicles Symposium*, pages 825–830. IEEE, 2017.

[23] Alexandros Nikou, Dimitris Boskos, Jana Tumova, and Dimos V Dimarogonas. On the timed temporal logic planning of coupled multi-agent systems. *Automatica*, 97:339–345, 2018.

[24] Xiao Li, Zachary Serlin, Guang Yang, and Calin Belta. A formal methods approach to interpretable reinforcement learning for robotic planning. *Science Robotics*, 4(37), 2019.

[25] Maxime Bouton, Jesper Karlsson, Alireza Nakhaei, Kikuo Fujimura, Mykel J Kochenderfer, and Jana Tumova. Reinforcement learning with probabilistic guarantees for autonomous driving. *arXiv preprint arXiv:1904.07189*, 2019.

[26] George Rupert Mason, Radu Constantin Calinescu, Daniel Kudenko, and Alec Banks. Assured reinforcement learning with formally verified abstract policies. In *ICAART*, 2017.

[27] Razieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. Bounded rational search for on-the-fly model checking of ltl properties. In *FSE*, pages 292–307. Springer, 2009.