Mälardalen University Press Dissertations
No. 359

# FORMAL METHODS FOR SCALABLE SYNTHESIS AND VERIFICATION OF AUTONOMOUS SYSTEMS

## MISSION PLANNING AND COLLISION AVOIDANCE

Rong Gu

2022

School of Innovation, Design and Engineering

ISBN 978-91-7485-552-4
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 359

# FORMAL METHODS FOR SCALABLE SYNTHESIS AND VERIFICATION OF AUTONOMOUS SYSTEMS
MISSION PLANNING AND COLLISION AVOIDANCE

Rong Gu

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras onsdagen den 15 juni 2022, 13.00 i Gamma & online, Mälardalens universitet, Västerås.

Fakultetsopponent: Professor Rajeev Alur, University of Pennsylvania

Akademin för innovation, design och teknik

## Abstract

Autonomous systems (a.k.a., agents) are often designed to move and execute tasks, without or with little human intervention. As the agents are often involved in safety- or mission-critical scenarios, ensuring the correctness of mission planning (i.e., path finding and task scheduling) and collision avoidance is crucial for such systems. However, traditional verification approaches, such as testing, are not sufficient to provide such assurance.

Formal methods such as model checking are well known for their rigorous verification based on mathematical models and logic rules, which provide guarantees of the absence of errors in system models. However, employing them entails tackling many challenges such as the complicated formal modeling and the scalability of the algorithmic methods. Additionally, the mission planning concerns the static and predictable factors in the working environment of the agents, such as stationary obstacles and predefined tasks, whereas the collision avoidance focuses on the dynamic and unpredictable factors, such as pedestrians. Consequently, certain questions arise in this context: (i) How can formal methods be applied in providing correctness-guaranteed solutions within a holistic framework that handles both the static mission planning and the dynamic collision avoidance?, and (ii) When the methods for realizing the agents' artificial intelligence, such as machine learning, involve large amounts of data, how to improve the scalability of formal methods when verifying the results of such methods? In this dissertation, we offer answers to the questions by developing solutions in form of new frameworks and algorithms targeting the mentioned problems, implementing the solutions in software tools, and evaluating their performance on real-world applications.

We propose a two-layer framework for formal modeling and verification of agents. The framework separates the discrete mission planning from the continuous movement of agents, which is needed for collision avoidance verification. Additionally, different formal modeling and verification techniques are adopted in the two layers of the framework respectively.

For mission planning, we design two types of tool-supported approaches, one based on graphic search, and one based on learning. The former is sound and complete, and supported by the tools UPPAAL and UPPAAL TIGA. However, the graphic-search approach is not scalable for large numbers of agents. The learning-based solution complements the graphic-search one, by handling more agents, being supported by UPPAAL STRATEGO. As a trade-off, the learning-based method is sound but not complete.

For the verification of collision avoidance, we propose two solutions, the first one based on statistical model checking in UPPAAL SMC, and the second one based on the symbolic model checking of UPPAAL STRATEGO. In the second solution, we transform the hybrid agent models, whose verification is undecidable, into a conservative over-approximation as a discrete-time model whose model checking is decidable. These results are proven as theorems in the dissertation.

To support our methods, we develop a toolset named MALTA that enables the automation of model construction and mission planning, and provides a visualization of environment configuration and the resulting mission plans. By using MALTA, we experiment with our novel methods in an industrial use case: an autonomous quarry. The experiment results demonstrate the advantages and weaknesses of different methods used in different types of environments, as well as the applicability of our methods and tool in complex systems.

ISBN 978-91-7485-552-4
ISSN 1651-4238

# Abstract

*Autonomous systems* (a.k.a., agents) are often designed to move and execute tasks, without or with little human intervention. As the agents are often involved in safety- or mission-critical scenarios, ensuring the correctness of mission planning (i.e., path finding and task scheduling) and collision avoidance is crucial for such systems. However, traditional verification approaches, such as testing, are not sufficient to provide such assurance.

Formal methods such as model checking are well known for their rigorous verification based on mathematical models and logic rules, which provide guarantees of the absence of errors in system models. However, employing them entails tackling many challenges such as the complicated formal modeling and the scalability of the algorithmic methods. Additionally, the mission planning concerns the static and predictable factors in the working environment of the agents, such as stationary obstacles and predefined tasks, whereas the collision avoidance focuses on the dynamic and unpredictable factors, such as pedestrians. Consequently, certain questions arise in this context: (i) How can formal methods be applied in providing correctness-guaranteed solutions within a holistic framework that handles both the static mission planning and the dynamic collision avoidance?, and (ii) When the methods for realizing the agents' artificial intelligence, such as machine learning, involve large amounts of data, how to improve the scalability of formal methods when verifying the results of such methods? In this dissertation, we offer answers to the questions by developing solutions in form of new frameworks and algorithms targeting the mentioned problems, implementing the solutions in software tools, and evaluating their performance on real-world applications.

We propose a *two-layer framework* for formal modeling and verification of agents. The framework separates the discrete mission planning from the continuous movement of agents, which is needed for collision avoidance verification. Additionally, different formal modeling and verification techniques are adopted in the two layers of the framework respectively.

For mission planning, we design two types of tool-supported approaches, one based on graphic search, and one based on learning. The former is sound and complete, and supported by the tools UPPAAL and UPPAAL TIGA. However, the graphic-search approach is not scalable for large numbers of agents. The learning-based solution complements the graphic-search one, by handling more agents, being supported by UPPAAL STRATEGO. As a trade-off, the learning-based method is sound but not complete.

For the verification of collision avoidance, we propose two solutions, the first one based on statistical model checking in UPPAAL SMC, and second one based on the symbolic model checking of UPPAAL STRATEGO. In the second solution, we transform the hybrid agent models, whose verification is undecidable, into a conservative over-approximation as a discrete-time model whose model checking is decidable. These results are proven as theorems in the dissertation.

To support our methods, we develop a toolset named *MALTA* that enables the automation of model construction and mission planning, and provides a visualization of environment configuration and the resulting mission plans. By using *MALTA*, we experiment with our novel methods in an industrial use case: an autonomous quarry. The experiment results demonstrate the advantages and weaknesses of different methods used in different types of environments, as well as the applicability of our methods and tool in complex systems.

# Sammanfattning

Autonoma system (även kallade agenter) är ofta designade för att förflytta sig och utföra uppdrag, helt utan eller med lite mänskligt ingripande. Eftersom agenter ofta är involverade i säkerhets- eller uppdragskritiska scenarion, är det avgörande att säkerställa korrektheten av uppdragsplaneringen (dvs. planering av färdväg och schemaläggning av uppgifter) hos sådana system. Men traditionella verifieringsmetoder, såsom testning, är inte tillräckliga för att ge sådan försäkran.

Formella metoder, exempelvis modellkontroll ("model checking"), är välkända för sin rigorösa algoritmiska verifiering baserad på matematiska modeller och logiska regler, vilken ger garantier för frånvaron av fel i systemmodeller. Att använda dem innebär dock att hantera många utmaningar såsom den komplicerade formella modelleringen och skalbarheten hos de algoritmiska metoderna. Dessutom så berör uppdragsplaneringen de statiska och förutsägbara faktorerna i agenternas arbetsmiljö, såsom stationära hinder och fördefinierade uppgifter, medan kollisionsundvikandet fokuserar på de dynamiska och oförutsägbara faktorerna, såsom fotgängare. Följaktligen uppstår frågor i detta sammanhang: (i) Hur kan formella metoder tillämpas för att tillhandahålla korrekthetsgaranterade lösningar inom ett holistiskt ramverk som hanterar både den statiska uppdragsplaneringen och det dynamiska kollisionsundvikandet?, och (ii) När algoritmerna för att realisera agenternas artificiella intelligens (AI), såsom maskininlärning, involverar stora mängder data, hur kan man förbättra skalbarheten av formella metoder när man verifierar resultaten av sådana algoritmer? I denna avhandling erbjuder vi svar på dessa frågor genom att utveckla lösningar i form av nya ramverk och algoritmer som riktar in sig på de nämnda problemen, implementera lösningarna i mjukvaruverktyg och utvärdera deras prestanda på verkliga applikationer.

Vi föreslår ett ramverk i två lager för formell modellering och verifiering av agenter. Ramverket skiljer den diskreta uppdragsplaneringen från den kontinuerliga rörelsen av agenter, som behövs för att verifiera kollisionsundvikande. Dessutom så används olika formella modellerings- och verifieringstekniker i ramverkets två lager.

För uppdragsplanering så designar vi två typer av verktygsstödda tillvägagångssätt, ett baserat på grafisk sökning och ett baserad på lärande. Det förstnämnda tillvägagångssättet är sunt och fullständigt, och stöds av verktygen UPPAAL och UPPAAL TIGA. Men, det är dock inte skalbart för ett stort antal agenter. Det lärningsbaserade tillvägagångssättet kompletterar det grafisk söknings-baserade, genom att hantera flera agenter, med stöd av UPPAAL STRATEGO. Som en avvägning, så är den lärningsbaserade metoden sund men inte fullständig.

För verifiering av kollisionsundvikande presenterar vi två lösningar, den första baserad på statistisk modellkontroll i UPPAAL SMC, och den andra baserad på den symboliska modellkontroll som finns i UPPAAL STRATEGO. I den andra lösningen transformerar vi hybridmodellen av agenter, vars verifiering är oavgörbar, till en konservativ överapproximation i form av en diskret tids-modell, vars modellkontroll är avgörbar. Dessa resultat bevisas som satser i avhandlingen.

För att underlätta tillämpningen av våra metoder i industriella system, utvecklar vi en verktygsuppsättning, MALTA, vilken möjliggör automatisering av modellkonstruktion och uppdragsplanering och ger en visualisering av miljökonfiguration och de resulterande uppdragsplanerna. Genom att använda MALTA utvärderar vi våra nya metoder på ett industriellt användningsfall: ett autonomt stenbrott. Utvärderingsresultaten visar fördelarna och svagheterna med olika metoder som används i olika typer av miljöer, och visar användbarheten av våra metoder och verktyg i komplexa system.

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

In September 2006, I was taking my first class in discrete mathematics. The professor asked us a question: *how can you prove the correctness of your software program?* I did not know the answer but since then, I started my journey of pursuing the answer to that question. After graduating from Xi'an Jiaotong University in 2013, I became an embedded software developer at an avionic institute, where I participated in several projects of designing control software of safety-critical systems, such as aircraft. That question, how to prove the correctness of software programs, had always been remaining in my mind. Testing and simulation are good for finding bugs, but how can I ensure that my programs are free of bugs? Curiosity drove me to apply for a doctoral student at the age of 29, a year earlier than what we Chinese call 而立 (*er-li*), which means a man should finish his study and be independent at that time.

After five years of study, can I answer that question? Yes and No. On one hand, I have been learning and using formal methods to prove the correctness of autonomous systems in my doctoral study. I am fascinated by formal modeling, model checking, and theorem proving, which are mathematically elegant yet practically useful. On the other hand, the more I study the more I know that there are still many unsolved problems in this field, which motivate me to carry on researching with the attitude of a student in the future.

During these five years, things were not always easy, sometimes even tough. Luckily, I have a lovely family behind me. My wife is always there for me. We took good care of each other and went through some difficult times altogether. 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 that 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 would like to thank my supervisors, Associate Professor Cristina Seceleanu, Professor Kristina Lundqvist, Senior Lecturer Eduard Enoiu, and former supervisor 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 in tolerating all my faults and waywardness, and patience in teaching me everything repetitively. Your consistent enthusiasm for work and life is and will always be my light for the path. Many thanks to Edi and Raluca. Your advice on how to do research, write papers and give presentations has been very helpful to me. I really enjoy the time when we work together and walk your dogs 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.

I would like to express my deep gratitude to the faculty examiner, Professor Rajeev Alur, and the grading committee members: Professor Kim Larsen, Professor Jan Křetínský, and Associate Professor Jana Tumova 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 dissertation.

I would like to thank Professor Hans Hansson for reviewing the proposal and the initial version of the dissertation, and Dr Peter Backeman for the help with the Swedish abstract.

I would like to express my appreciation to my colleagues at Mälardalen University (MDU), Lecturer Afshin Ameri, Professor Mats Björkman, and Professor Mikael Sjödin for providing me the chance to give lectures to our students. Many thanks to Senior Lecturer Baran Çürüklü at MDU, Dr Eduard Baranov and Professor Axel Legay at UCLouvain, and Assistant Professor Peter Jensen at Aalborg University. I have learned a lot in our collaboration for solving the synthesis problem. The pandemic of Covid-19 has stopped the traveling, but our online discussion and communication went successfully and fruitfully. I would like to thank my friends, Asha, Damir, Nesredin, Predrag, and Simin, etc., for being supportive and giving me advice to be a good doctoral student. Although we may not see each other as often as before, it is like what the beautiful Chinese poem says, a bosom friend afar brings distance near.

Rong Gu
Västerås, March, 2022

# List of Publications

## Papers Included in the Dissertation[1]

**Paper A** *Towards a Two-Layer Framework for Verifying Autonomous Vehicles.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Published in Proceedings of the $11^{\text{th}}$ Annual NASA Formal Methods Symposium, LNCS vol.11460, pp.186-203, Springer, Houston, USA, 2019.

**Paper B** *Verifiable Strategy Synthesis for Multiple Autonomous Agents: A Scalable Approach.* Rong Gu, Peter G. Jensen, Danny B. Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Published in Journal of Software Tools for Technology Transfer (STTT), Special Issue: FMICS 2019/2020, pp.1-20, Springer, 2022.

**Paper C** *Synthesis and Verification of Mission Plans for Multiple Autonomous Agents under Complex Road Conditions.* Rong Gu, Eduard Baranov, Afshin Ameri, Eduard Enoiu, Baran Cürüklü, Cristina Seceleanu, Axel Legay, and Kristina Lundqvist. Submitted to Transactions on Software Engineering and Methodology (TOSEM), ACM, 2022.

**Paper D** *Probabilistic Mission Planning and Analysis for Multi-agent Systems.* Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Published in Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), LNCS vol.12476, pp.350-367, Springer, Rhodes, Greece, 2021.

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

**Paper E** *Correctness-Guaranteed Strategy Synthesis and Compression for Multi-Agent Autonomous Systems.* Rong Gu, Peter G. Jensen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Submitted to Journal of Science of Computer Programming (SCP), Elsevier, 2022.

**Paper F** *Model Checking Collision Avoidance of Nonlinear Autonomous Vehicle Models.* Rong Gu, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Published in Proceedings of the 24th International Symposium on Formal Methods (FM). LNCS vol.13047, pp.676-694, Springer, online, 2021.

## Papers Related to, but not Included in the Dissertation

**1.** *Formal Verification of an Autonomous Wheel Loader by Model Checking.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Published in Proceedings of the 6th Conference on Formal Methods in Software Engineering (FormaliSE), pp.74-83, ACM, 2018.

**2.** *TAMAA: UPPAAL-based Mission Planning for Autonomous Agents.* Rong Gu, Eduard Enoiu, and Cristina Seceleanu. Published in Proceedings of the 35th Symposium on Applied Computing (SAC), pp.1624-1633, ACM, 2020.

**3.** *Verifiable and Scalable Mission-Plan Synthesis for Multiple Autonomous Agents.* Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Published in Proceedings of the 25th International Conference on Formal Methods for Industrial Critical Systems (FMICS). LNCS vol.12327, pp.73-92, Springer, Vienna, Austria, 2020. **Best Paper Award**.

# Contents

I Dissertation 1

1 Introduction 3
- 1.1 A Two-Layer Framework for Modeling and Verifying Autonomous Systems . . . 6
- 1.2 Mission Planning for Multi-Agent Autonomous Systems . 6
- 1.3 Reach-Avoid Verification of Nonlinear Agents . . . . . . . 9
- 1.4 Dissertation Overview . . . . . . . 10

2 Preliminaries 17
- 2.1 Formal Verification and Synthesis . . . . . . . 17
  - 2.1.1 Formal Verification . . . . . . . 17
  - 2.1.2 Synthesis . . . . . . . 19
- 2.2 Modeling Languages and Tools . . . . . . . 19
  - 2.2.1 Timed Automata and UPPAAL . . . . . . . 20
  - 2.2.2 Hybrid Automata and UPPAAL SMC . . . . . . . 22
  - 2.2.3 Timed Games and Strategies . . . . . . . 24
- 2.3 Path Finding and Collision Avoidance . . . . . . . 25
  - 2.3.1 Theta* Algorithm . . . . . . . 25
  - 2.3.2 Collision Avoidance based on Dipole Flow Fields . 26
- 2.4 Reinforcement Learning . . . . . . . 26

3 Research Problem 29
- 3.1 Problem Description . . . . . . . 29
- 3.2 Research Goals . . . . . . . 31

4 Research Methods 35

**5 Dissertation Contributions** **37**

5.1 A Two-Layer Framework for Modeling and Verifying Autonomous Systems . . . . . . . . . . . . . . . . . . . . . . 37

5.1.1 Overall Description of the Framework . . . . . . . 37

5.1.2 Communication between the Two Layers . . . . . . 39

5.2 Mission Planning in Different Environments . . . . . . . . 41

5.2.1 Mission Planning in Deterministic Environments . 45

5.2.2 Mission Planning in Non-deterministic Environments . . . . . . . . . . . . . . . . . . . . . . . . 47

5.2.3 Mission Planning in Stochastic Environments . . . 52

5.2.4 Correctness-Guaranteed Mission Plan Compression 52

5.3 Model-Checking Reach-Avoid Requirement of Nonlinear Agents . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

5.3.1 Modeling Nonlinear Agents . . . . . . . . . . . . . 54

5.3.2 Solution A: Statistical Model Checking . . . . . . . 56

5.3.3 Solution B: Exhaustive Model Checking . . . . . . 57

5.4 Validation on Industrial Systems . . . . . . . . . . . . . . 60

5.5 Research Goals Revisited . . . . . . . . . . . . . . . . . . 64

**6 Related Work** **67**

6.1 Multi-Layer Frameworks for Agent Design and Verification 67

6.2 Mission Planning of Agents . . . . . . . . . . . . . . . . . 68

6.3 Verification of Agents . . . . . . . . . . . . . . . . . . . . 70

**7 Conclusions and Future Work** **73**

7.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . 73

7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . 75

**Bibliography** **77**

**II Included Papers** **89**

**8 Paper A: Towards a Two-layer Framework for Verifying Autonomous Vehicles** **91**

8.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 93

8.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . 94

8.2.1 Hybrid Automata and UPPAAL SMC . . . . . . 94

8.2.2 Theta* Algorithm . . . . . . . . . . . . . . . . . . 96

8.2.3 Dipole Flow Field for Collision Avoidance . . . . . 96

8.3 Use Case: Autonomous Wheel Loader . . . . . . . . . . . 97
8.4 A Two-level Framework for Planning and Verifying Autonomous Vehicles . . . . . . . . . . . . . . . . . . . . . . 99
8.5 Pattern-based Modeling of the Dynamic Layer . . . . . . 101
8.5.1 Patterns for the Execution Unit . . . . . . . . . . . 101
8.5.2 Patterns for the Control Unit . . . . . . . . . . . . 102
8.5.3 Encoding the Control Unit Patterns as Hybrid Automata . . . . . . . . . . . . . . . . . . . . . . . . 104
8.6 Use Case Revisited: Applying Our Method on AWL . . . . . . . . . . . . . . . . . . . . . . . 106
8.6.1 Formal Model of the Control Unit . . . . . . . . . 107
8.6.2 Statistical Model Checking of the AWL Formal Model . . . . . . . . . . . . . . . . . . . . . . . . . 108
8.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 111
8.8 Conclusions and future work . . . . . . . . . . . . . . . . 112
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

**9 Paper B: Verifiable Strategy Synthesis for Multiple Autonomous Agents: A Scalable Approach 117**
9.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 119
9.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . 122
9.2.1 Timed Automata and Timed Games . . . . . . . . 122
9.2.2 Stochastic Timed Games and Stochastic Strategies 124
9.2.3 UPPAAL, UPPAAL TIGA, and UPPAAL STRATEGO . . . . . . . . . . . . . . . . . . . . . . . . . 125
9.2.4 Reinforcement Learning . . . . . . . . . . . . . . . 126
9.3 Problem Description and Analysis . . . . . . . . . . . . . 127
9.3.1 An Industrial Case Study: The Autonomous Quarry 127
9.3.2 Problem Analysis . . . . . . . . . . . . . . . . . . . 128
9.3.3 Non-determinism and Scalability of Mission Planning . . . . . . . . . . . . . . . . . . . . . . . 129
9.4 Overall Description of the Solutions . . . . . . . . . . . . 130
9.5 Solution 1: Game-Theoretic Synthesis . . . . . . . . . . . 132
9.5.1 Overall Description of TAMAA . . . . . . . . . . . 132
9.5.2 Mission-Plan Synthesis by TAMAA . . . . . . . . 132
9.5.3 Synthesizing Strategies in UPPAAL TIGA . . . . 137
9.6 Solution 2: Simulation-Based Synthesis . . . . . . . . . . . 140
9.6.1 State-Space Exploration of TAMAA . . . . . . . . 140
9.6.2 Learning Strategies . . . . . . . . . . . . . . . . . . 140

9.7 Tool Support . . . . . . . . . . . . . . . . . . . . . . . . . 149
9.7.1 Integration of Task Scheduler and UPPAAL STRATEGO . . . . . . . . . . . . . . . . . . . . . . . . 151
9.8 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . 152
9.8.1 Design of Experiments . . . . . . . . . . . . . . . . . 152
9.8.2 Results of Experiments . . . . . . . . . . . . . . . . 152
9.8.3 Discussion of the Experimental Results . . . . . . 155
9.9 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 157
9.10 Conclusions and Future Work . . . . . . . . . . . . . . . . . 159
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . 161

**10 Paper C: Synthesis and Verification of Mission Plans for Multiple Autonomous Agents under Complex Road Conditions 167**
10.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 169
10.2 An Industrial Case Study: The Autonomous Quarry . . . 173
10.3 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . 176
10.3.1 Timed Automata and UPPAAL . . . . . . . . . . . 177
10.3.2 Devices for Assisted Living (DALI) . . . . . . . . . 179
10.4 Mission Planning Methodology . . . . . . . . . . . . . . . 180
10.4.1 Improved DALI for Path Planning . . . . . . . . . 183
10.4.2 TAMAA for Task Scheduling . . . . . . . . . . . . 187
10.4.3 Mission Planning with DALI and TAMAA . . . . 198
10.5 Description of the Tool . . . . . . . . . . . . . . . . . . . . 201
10.5.1 Overall Description . . . . . . . . . . . . . . . . . . 201
10.5.2 Mission Management Tool . . . . . . . . . . . . . . 203
10.5.3 Environmental Configuration with MMT . . . . . 204
10.5.4 Mission Plan Demonstration in MMT . . . . . . . 206
10.6 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 207
10.6.1 Methodology . . . . . . . . . . . . . . . . . . . . . . 208
10.6.2 Comparison of DALI optimizations . . . . . . . . 210
10.6.3 Comparison of A* and DALI . . . . . . . . . . . . 210
10.6.4 Evaluation of the Approach with Heat Areas and Temporary Obstacles . . . . . . . . . . . . . . . . . 213
10.6.5 Results for Multiple Agents . . . . . . . . . . . . . 215
10.7 Adaptability of MALTA: a Special Industrial Use Case . . 219
10.7.1 Adjustments of the Models . . . . . . . . . . . . . 221
10.7.2 Additional Adaptation of Queries and Models . . . 223
10.7.3 Synthesis Results . . . . . . . . . . . . . . . . . . . 225

10.8 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 226
10.9 Conclusions and Future Work . . . . . . . . . . . . . . . . 229
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . 231

**11 Paper D: Probabilistic Mission Planning and Analysis for Multi-agent Systems 237**
11.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 239
11.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . 241
11.2.1 Stochastic Timed Automata and UPPAAL SMC . 241
11.2.2 Reinforcement Learning . . . . . . . . . . . . . . . 242
11.2.3 A Two-Layer Framework for Formal Modelling and Verification of Autonomous Agents . . . . . . . . . 243
11.3 Problem Description . . . . . . . . . . . . . . . . . . . . . 244
11.4 Mission Planning Based on Reinforcement Learning and Stochastic Timed Automata . . . . . . . . . . . . . . . . . 246
11.4.1 MCRL: Combining Model Checking and Reinforcement Learning for Mission Planning . . . . . . . . 247
11.4.2 Stochastic Timed Automata for MCRL . . . . . . 249
11.5 Statistical Verification and Analysis of the Use Case: an Autonomous Quarry . . . . . . . . . . . . . . . . . . . . . 253
11.5.1 Mission Plan Synthesis . . . . . . . . . . . . . . . . 254
11.5.2 Bottleneck Analysis . . . . . . . . . . . . . . . . . 255
11.5.3 Travelling Timed Estimation and Re-Planning . . 256
11.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 259
11.7 Conclusions and Future Work . . . . . . . . . . . . . . . . 260
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . 261

**12 Paper E: Correctness-Guaranteed Strategy Synthesis and Compression for Multi-Agent Autonomous Systems 265**
12.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 267
12.2 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . 269
12.2.1 UPPAAL Timed Automata . . . . . . . . . . . . . 269
12.2.2 Timed Games . . . . . . . . . . . . . . . . . . . . . 270
12.2.3 Model Checking and Temporal Properties . . . . . 272
12.2.4 Reinforcement Learning . . . . . . . . . . . . . . . 273
12.3 Problem Description . . . . . . . . . . . . . . . . . . . . . 273
12.3.1 Overall Description . . . . . . . . . . . . . . . . . . 274
12.3.2 Challenges of Solving the Planning Problem . . . . 274
12.3.3 A Motivating Example . . . . . . . . . . . . . . . . 277

12.4 Strategy Synthesis, Verification and Compression . . . . . 278
12.4.1 Overall Workflow of MoCReL . . . . . . . . . . . . 278
12.4.2 Modeling of MAS . . . . . . . . . . . . . . . . . . . 279
12.4.3 Partial State-Space Observation . . . . . . . . . . 283
12.4.4 Key Techniques of MoCReL . . . . . . . . . . . . . 284
12.5 Experimental Evaluation . . . . . . . . . . . . . . . . . . . 294
12.5.1 Use Case Description . . . . . . . . . . . . . . . . . 295
12.5.2 Experiment Design . . . . . . . . . . . . . . . . . . 296
12.5.3 Experiment Results . . . . . . . . . . . . . . . . . 297
12.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 300
12.7 Conclusions and Future Work . . . . . . . . . . . . . . . . 301
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . 303

**13 Paper F: Model Checking Collision Avoidance of Nonlinear Autonomous Vehicle Models 309**
13.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 311
13.2 Problem Description . . . . . . . . . . . . . . . . . . . . . 313
13.3 Definitions and Verification Reduction Theorems . . . . . 315
13.3.1 Definitions of Maps, Agent States, and Trajectories 316
13.3.2 Collision-Avoidance Verification Reduction . . . . . 319
13.3.3 Discretization of Trajectories . . . . . . . . . . . . 320
13.4 Verification Approach and Tool Support . . . . . . . . . . 323
13.4.1 General Description of the Approach . . . . . . . . 324
13.4.2 Design of the UTA Templates and CTL Properties 325
13.4.3 Reduction of the State Space of the UTA Model . 326
13.5 Experimental Evaluation . . . . . . . . . . . . . . . . . . . 327
13.5.1 The Collision-Avoidance Algorithm to be Verified . 328
13.5.2 Verification Results . . . . . . . . . . . . . . . . . . 328
13.6 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 331
13.7 Conclusions and Future Work . . . . . . . . . . . . . . . . 332
Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . 333

**A 337**
A.1 Algorithm of Synthesis . . . . . . . . . . . . . . . . . . . . 337
A.2 Algorithm of Verification and Labeling . . . . . . . . . . . 337
A.3 Templates of the TG models . . . . . . . . . . . . . . . . 340
A.4 Overview of the External Library of MoCReL . . . . . . . 342

# I

# Dissertation

# Chapter 1

# Introduction

Autonomous systems are systems that can move and execute certain tasks without or with little human intervention. Such systems, which are referred to as autonomous agents, or shortly agents [1], are often designed to move within a confined environment and accomplish a common mission, sometimes collectively. To achieve this design goal, two crucial functions of autonomy need to be designed: mission planning or mission-plan synthesis (that includes path finding and task scheduling), and collision avoidance. The first function aims to generate mission plans for agents to move without hitting static obstacles, and accomplish certain tasks with respect to certain requirements, e.g., a robotic mailman delivering all letters within two hours. The second function is about avoiding dynamic obstacles, e.g., humans, when agents are executing the mission plans. Additionally, agents are often involved in safety- or mission-critical scenarios where malfunctions of the systems can result in casualties and profound property damage. According to an analysis of traffic accidents that occurred in the US state of California from January 2015 to December 2017 [2], the risk of a traffic accident with causalities involving autonomous vehicles is almost eight times higher than that of human-driven vehicles. Therefore, a thorough analysis of the agents' autonomous functionality is crucial for obtaining a guarantee of their safe operation. In cases that the agents operate in industrial settings, such as in construction sites, besides safe operation they are also required to guarantee a certain productivity. The uncertainties in the environment may hinder the agents from fulfilling such requirements. For instance, pedestrians that appear arbitrarily in a site may cause the agents to slow down or deviate from their original paths, and thus failing to accomplish their mission in time.

Traditional approaches of verification, such as simulation and prototype testing, can be expensive and sometimes not sufficient for assuring a system that interacts autonomously with an unpredictable environment that assumes the presence of humans and varying site conditions [3]. If one considers an example of a fleet of agents, guaranteeing less than one catastrophic failure per hour needs more than 10 billion test cases [4]. Additionally, the traditional approaches of verification are often applied after the system's design phase, which makes it expensive to correct mistakes that could have been detected and removed at or before that phase already, such as the system's specification phase.

*Formal methods* are mathematically rigorous techniques for system specification, design, validation, and verification [5]. Therefore, *formal verification* can be applied for assuring such autonomous systems at design time, as a complement to the traditional verification techniques. *Model checking* is a formal method that explores the state space of a model and checks if it satisfies certain temporal logic properties [6], such as always executing task A, and then B, followed by C, possibly within a given time. When the state-space exploration is exhaustive, model checking provides a guarantee that a system model satisfies a certain requirement, which might not be possible for traditional verification techniques. Besides verification, tool-supported formal methods also provide a possibility of automatically constructing mission plans with a correctness guarantee, that is, *synthesizing* mission plans that are correct-by-construction [7].

Several related studies [8, 9, 10, 11, 12] on motion planning and the verification of agents have inspired us to address the problems of mission planning and collision avoidance verification by leveraging state-of-the-art formal methods. However, applying formal methods on such complex systems has many challenges:

(i) The inherent difference between mission planning and dynamic collision avoidance. The former aspect does not concern the kinematic features of the agents as it only focuses on computing a path plan towards the destination, and a schedule carrying out certain tasks, in a certain order, at given positions called milestones, within prescribed amounts of time. However, verifying dynamic collision avoidance requires a continuous environment, in which the kinematics of the agents can be captured. Therefore, one needs a modeling and verification solution that decouples the discrete part from the continuous part in order to provide a separation of concerns.

(ii) Composing a large number of agents within a model gives rise to a dramatically large state space, which makes the problem of mission-plan synthesis and verification computationally challenging [13, 14]. Additionally, different types of environments require different synthesis methods, such as collaborative environments being under the full control of agents and non-deterministic ones acting arbitrarily. However, the resulting mission plans are supposed to let the agents accomplish their mission regardless of how the environment acts. Hence, our synthesis methodology must be scalable to deal with a large number of agents and be able to function in different types of environments.

(iii) The combination of the discrete control of an agent's behavior, and its continuous motions gives rise to a hybrid system model (e.g., captured via hybrid automata [15]), on which exhaustive model checking cannot be applied as such, since it is undecidable [16, 17].

(iv) Few of the related studies consider timing requirements for mission-plan synthesis, which are of high concern in industrial applications. Our verification techniques must be applicable for real-world cases, such as industrial systems.

In this dissertation, we address the above challenges by the following contributions:

- A two-layer framework for modeling and verification of agents - Challenge (i) (Section 5.1).
- A scalable methodology for mission planning in different types of environments - Challenge (ii) (Section 5.2).
- A methodology for reach-avoid verification considering the nonlinear trajectories of agents - Challenge (iii) (Section 5.3).
- Tools that facilitate the use of our methodology in industrial applications - Challenge (iv) (Section 5.4).

Next, we give a more detailed introduction of the four parts of the dissertation's contributions, followed by an overview of the dissertation.

## 1.1 A Two-Layer Framework for Modeling and Verifying Autonomous Systems

In this dissertation, we propose a *two-layer framework* for modeling and verification of agents consisting of a *static* and a *dynamic* layer, respectively, with data being exchanged between them [18]. The *static layer* is responsible for mission planning for the agents according to possibly incomplete information of the environment. In this layer, known static obstacles are assumed, together with milestones representing points of task operation of the agents. The *dynamic layer* is dedicated to simulating and verifying the agents that follow the reference path from the starting point to the destination, generated by the static layer, while considering unforeseen static and moving obstacles. The structure of the framework separates the static high-level mission planning from the dynamic collision avoidance, thus providing a separation of concerns for the system's modeling and verification. The communication between these two layers enables re-planning when unforeseen obstacles are detected in the dynamic layer and the static layer needs to update the mission plan accordingly.

## 1.2 Mission Planning for Multi-Agent Autonomous Systems

Mission planning for *Multi-Agent Autonomous Systems* (*MAS*) can be categorized into three types of problems according to the environment types, respectively: 1) a *deterministic* environment where agents control which actions start and finish, and when; 2) a *stochastic* environment where agents decide only which actions start and when, whereas the environment chooses each action's ending time stochastically; 3) a *non-deterministic* environment that is similar to a stochastic one, with a difference that the respective ending time of any action is decided by the environment non-deterministically. Next, we introduce the solutions for each type of environment, respectively.

*Solution for environments of type 1).* In a deterministic environment, a mission plan consists of the starting and ending time of the agents' actions, respectively, that is, moving to a milestone or starting a task. We model the mission-planning problem in such environments as a 1-player game where the MAS represent the only player in the game. The goal of winning the game is to order the agents' actions so that they visit the milestones in a certain order and finish their tasks in the shortest time while satisfying the constrains of tasks, e.g., always finishing task A before task B starts. We use *timed automata* (TA) [19] as the modeling language and the model checker UPPAAL [20] to solve the game, that is, synthesize a plan (a.k.a., winning strategy) for agents to win the game. In addition, the method is supported by our tool named *MALTA*[1] [21]. The tool provides a graphical user interface (GUI) for configuring the environment and tasks for the agents, and an extensible module of path finding and task scheduling. So far, *MALTA* supports three path-finding algorithms, A* [22], Theta* [23], and DALi [24], and one task-scheduling method that uses timed automata and UPPAAL [13].

*Solution for environments of type 2).* In a stochastic environment, the duration of actions is no longer a fixed number but a time interval between the best-case-execution time (BCET) and the worst-case-execution time (WCET). In other words, the time point of finishing an action is chosen by the environment stochastically between the action's BCET and WCET. For example, if completing an action is equally likely between its BCET and WCET, the uniform distribution of probabilities can be used to model the environment's stochastic decision of completing the action. Now, the mission-planning problem in such environments can be modeled as a $1\frac{1}{2}$-player game, in which one player, i.e., the MAS, behaves non-deterministically, and the environment is counted as $\frac{1}{2}$ of a player whose behavior is stochastic. The goal of winning the game is to choose the agents' starting time of actions according to the environment's stochastic reactions, so that the probability of visiting the milestones in the right order and finishing all the tasks within a certain time, while satisfying the task constraints, is higher than a required number.

*Solution for environments of type 3).* In a non-deterministic environment, the environment's behavior becomes non-deterministic, meaning that the actions are completed at arbitrary time points. Hence, the mission-planning problem now becomes a 2-player game, where both

[1] *MALTA* is published: https://github.com/rgu01/MALTA

the MAS and the environment are players of the game, who control their actions non-deterministically. Now the goal of winning the game is to choose the agents' starting time of actions, respectively, so that they can visit the milestones in the right order and finish all the tasks within a certain time while satisfying the task constraints regardless of how the environment reacts.

To solve the 2-player games captured by formal models called timed games (TG), adapted from the TA models in *MALTA*, we use the so-called UPPAAL TIGA tool [25, 26]. The problem that UPPAAL TIGA can solve limits the number of agents to five, because the linear growth of the agents' number increases the TG's state space exponentially, which causes the infamous state-space-explosion problem of model checking [27]. To deal with this limit, we design a novel synthesis approach that combines model checking with reinforcement learning [28], called MCRL[2] [29]. Our new approach is able to deal with more than five agents within a reasonable computation time. As an improvement of MCRL, we further develop the approach by integrating it into the tool called UPPAAL STRATEGO [30, 26]. The integration facilitates the modeling phase, accelerates the learning phase for synthesizing mission plans, and enables compressing the synthesized strategies while preserving their properties, such as "always eventually finishing all the tasks" [31].

We also show that MCRL can be used in solving $1\frac{1}{2}$-player games [32], and provide probabilistic results of verification by using statistical model checking [33]. The results expose the bottleneck of a strategy, which shows the positions where agents most likely wait the longest time, and enable the agents to re-calculate the mission plans considering moving obstacles, e.g., pedestrians, which are unpredictable and appear stochastically in the environment. The resulting mission plans are statistically optimal in the sense that they are most likely (i.e., with the highest probability) to be able to finish the agents' tasks in time.

Our method called *MoCReL*[3] is for compressing strategies produced by MCRL [31]. The compression can reduce the strategies' sizes down to 0.05% of the original ones while preserving their properties. This contribution is especially meaningful when the mission plans are supposed to be comprehensible by humans, or used in embedded systems with a limited memory space.

[2]MCRL: Model Checking + Reinforcement Learning
[3]MoCReL: Model-checked Compressed Reinforcement Learning

## 1.3 Reach-Avoid Verification of Nonlinear Agents

When agents start to execute the synthesized mission plans, they need to follow the planned paths and visit the milestones to carry out their tasks autonomously. The traveling now becomes continuous instead of the series of discrete steps in the mission plan. The agents may even encounter unforeseen obstacles, which can be also moving unpredictably. The manners of the moving obstacles' behavior form two types of environments: non-deterministic environments and stochastic environments. In both types of environments, the moving obstacles can appear at any moments, at any positions within the space, travel at any speeds and towards any directions within their capabilities. When the probabilities of when and where the obstacles appear and how they move are known, the environment is considered a stochastic one; otherwise, it is a non-deterministic environment. Whatever type the environment is, the agents are supposed to travel safely from their initial positions to destinations, respectively, without colliding with the static and moving obstacles. We call such a requirement the *reach-avoid* requirement, a term that is also used and studied in the literature [34, 35, 36].

To model the continuous movement of agents and moving obstacles, we use nonlinear hybrid automata [15], which reflect the nonlinear evolution of their continuous variables, e.g., positions, and the discrete state transitions, e.g., from stopped to moving. Unfortunately, model checking nonlinear hybrid automata is an undecidable problem [16, 17]. We use two methods to overcome this difficulty: (a) statistical model checking (SMC), and (b) symbolic model checking after a conservative transformation of the hybrid models. First, we employ statistical model checking supported by UPPAAL SMC [33] for verifying hybrid automata against the reach-avoid requirement [18]. Instead of exhaustively exploring the state space of the model, UPPAAL SMC simulates the model by the Monte-Carlo simulation and samples traces before checking whether the traces satisfy a certain property or not. The ones that satisfy the property are successful traces and the ratio of such successful traces among all samples is returned from the model checker as the probability of satisfaction of the property. In this way, the verification of hybrid automata is realized with a sacrifice of completeness, that is, quantitative answers (i.e., probabilities) replacing the qualitative answers (i.e., true/false).

When the environment becomes non-deterministic, the behavior of moving obstacles becomes non-deterministic, hence statistical model checking is not suitable here. To cope with the undecidable problem of model checking nonlinear hybrid automata, we conservatively transform the verification of an agent's continuous trajectory to the one of over-approximating discrete-time trajectory, and prove that, under certain assumptions, if the latter satisfies the reach-avoid requirement, the former must also satisfy it [37]. Based on this conclusion, we develop a method for model checking the reach-avoid requirement against TA models of agents and moving obstacles, whose model checking is decidable. The method uses the model checker in UPPAAL STRATEGO [30], as the tool supports calling external libraries, which provides a flexible way of embedding the collision avoidance algorithms into the TA. The model transformation does not sacrifice the level of assurance like in the case of employing statistical model checking, that is, if the verification returns *true*, the nonlinear hybrid models of agents are guaranteed to travel safely within the environment, and reach their destinations when facing any moving obstacles.

The two-layer framework and the corresponding novel approaches for mission plan synthesis and reach-avoid verification are evaluated via experiments on an industrial application, i.e., an autonomous quarry, provided by Volvo Construction Equipment in Sweden. The experimental evaluations show the performance of different mission-planning methods in different types of environments [26, 31]. Furthermore, we verify a novel algorithm of collision avoidance that is based on dipole flow fields [38] by using statistical model checking [18] and exhaustive model checking combined with our method of model transformation [37]. The methods reveal several problematic situations that benefit the algorithm designers to improve their method. We further demonstrate the absence of mistakes for an improved version of the algorithm in environments with one moving obstacle [37], which is impossible for traditional methods such as testing and simulation.

## 1.4 Dissertation Overview

This dissertation is divided into two parts. The first part is a summary of our research, including the preliminaries of this dissertation (Chapter 2), the problem formulation and our research goals (Chapter 3), the research methods applied in this dissertation (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 dissertation, listed as follows:

**Paper A** *Towards a Two-Layer Framework for Verifying Autonomous Vehicles.* Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. In *Proceedings of the 11$^{th}$ Annual NASA Formal Methods Symposium, LNCS volume 11460, pages 186-203, Springer, Houston, USA, 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 B** *Verifiable Strategy Synthesis for Multiple Autonomous Agents: A Scalable Approach.* Rong Gu, Peter G. Jensen, Danny B. Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. In *Journal of Software Tools for Technology Transfer (STTT), Special*

*Issue: FMICS 2019/2020, pages 1-20, Springer, 2022.*

**Abstract:** Path planning and task scheduling are two challenging problems in the design of multiple autonomous agents. Both problems can be solved by the use of exhaustive search techniques such as model checking and algorithmic game theory. However, model checking suffers from the infamous state-space explosion problem that makes it inefficient at solving the problems when the number of agents is large, which is often the case in realistic scenarios. In this paper, we propose a new version of our novel approach called MCRL that integrates model checking and reinforcement learning to alleviate this scalability limitation. We apply this new technique to synthesize path planning and task scheduling strategies for multiple autonomous agents. Our method is capable of handling a larger number of agents if compared to what is feasibly handled by the model-checking technique alone. Additionally, MCRL also guarantees the correctness of the synthesis results via post-verification. The method is implemented in UPPAAL STRATEGO and leverages our tool MALTA for model generation, such that one can use the method with less effort of model construction and a higher efficiency of learning than those of the original MCRL. We demonstrate the feasibility of our approach on an autonomous quarry industrial case study, and discuss the strengths and weaknesses of the methods.

**My contribution:** I was the primary driver of the paper, wrote most of the text. Peter Jensen and Danny Poulsen are external collaborators from Aalborg University, Denmark, who were responsible for the technical part of UPPAAL STRATEGO. The remaining authors provided valuable ideas and comments.

**Paper C** *Synthesis and Verification of Mission Plans for Multiple Autonomous Agents under Complex Road Conditions.* Rong Gu, Eduard Baranov, Afshin Ameri, Eduard Enoiu, Baran Cürüklü, Cristina Seceleanu, Axel Legay, and Kristina Lundqvist. Submitted to *Transactions on Software Engineering and Methodology (TOSEM), ACM, 2022.*

**Abstract:** Mission planning for multi-agent autonomous systems aims to generate feasible and optimal mission plans that satisfy the given requirements. In this article, we propose a mission-planning methodol-

ogy that combines (i) a path-planning algorithm for synthesizing path plans that are safe in environments with complex road conditions, and (ii) a task-scheduling method for synthesizing task plans that schedule the tasks in the right and fastest order, taking into account the planned paths. The task-scheduling method is based on model checking, which provides means of automatically generating task execution orders that satisfy the requirements and ensure the correctness and efficiency of the plans by construction. We implement our approach in a tool named MALTA, which offers a user-friendly GUI for configuring mission requirements, a module for path planning, an integration with the model checker UPPAAL, and functions for automatic generation of formal models, and parsing of the execution traces of models. Experiments with the tool demonstrate its applicability and performance in various configurations of an industrial case study of an autonomous quarry. We also show the adaptability of our tool by employing it on a special case of the industrial case study.

**My contribution:** I was the primary driver of the paper, wrote the majority of the text. Eduard Baranov designed of the path-finding algorithm called DALi that was integrated in our tool MALTA and wrote the technical part of DALi. Afshin Ameri and Baran Cürüklü were responsible for the front-end of MALTA, that is, MMT, and wrote the introduction of MMT. The remaining authors provided valuable ideas and comments.

**Paper D** *Probabilistic Mission Planning and Analysis for Multi-agent Systems.* Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. In Proceedings of the *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), LNCS volume 12476, pages 350-367, Springer, Rhodes, Greece, 2021.*

**Abstract:** Mission planning is one of the crucial problems in the design of autonomous Multi-Agent Systems (MAS), requiring the agents to calculate collision-free paths and efficiently schedule their tasks. The complexity of this problem greatly increases when the number of agents grows, as well as timing requirements and stochastic behavior of agents are considered. In this paper, we propose a novel method that integrates statistical model checking and reinforcement learning for mission planning within such context. Additionally, in order to synthesise mission plans that are statistically optimal, we employ hybrid automata to model the continuous movement of agents and moving obstacles, and estimate the possible delay of the agents' travelling time when facing unpredictable obstacles. We show the result of synthesising mission plans, analyze bottlenecks of the mission plans, and re-plan when pedestrians suddenly appear, by modeling and verifying a real industrial use case in UPPAAL SMC.

**My contribution:** I was the primary driver of the paper, wrote most of the text. All the remaining authors provided feedback and valuable ideas and comments.

**Paper E** *Correctness-Guaranteed Strategy Synthesis and Compression for Multi-Agent Autonomous Systems.* Rong Gu, Peter G. Jensen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Submitted to *Journal of Science of Computer Programming (SCP), Elsevier, 2022.*

**Abstract:** Planning is a critical function of multi-agent autonomous systems, which includes path-finding and task scheduling. Exhaustive search-based methods such as model checking and algorithmic game theory can solve simple instances of multi-agent planning. However, these methods suffer from state-space explosion when the number of agents is large. Learning-based methods can alleviate this problem, but lack a guarantee of correctness of the result. In this paper, we introduce MoCReL, a new version of our previously proposed method that combines model checking with reinforcement learning in solving the planning problem. The approach takes advantage of reinforcement learning to synthesize path plans and task schedules for large numbers of autonomous agents, and of model checking to verify the correctness of the synthesized strategies. Further, MoCReL can compress large strategies into smaller ones that have 0.05% of the original sizes, while preserving their correctness, which we show in this paper. MoCReL is integrated into a new version of UPPAAL STRATEGO that supports calling external libraries when running learning and verification of timed games models.

**My contribution:** I was the primary driver of the paper, wrote most of the text. Peter Jensen was the external collaborator from Aalborg University, Denmark, who was responsible for the technical part of UPPAAL STRATEGO. The remaining authors provided valuable comments.

**Paper F** *Model Checking Collision Avoidance of Nonlinear Autonomous Vehicle Models.* Rong Gu, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. In Proceedings of the *24th International Symposium On Formal Methods (FM), LNCS volume 13047, pages 676-694, Springer, online, 2021.*

**Abstract:** Autonomous vehicles are expected to be able to avoid static and moving obstacles automatically, along their way. However, most of the collision-avoidance functionality is not formally verified, which hinders ensuring such systems' safety. In this paper, we introduce formal definitions of the vehicle's movement and trajectory, based on hybrid transition systems. Since formally verifying hybrid systems algorithmically is undecidable, we reduce the verification of nonlinear vehicle models to verifying discrete-time vehicle models. Using this result, we propose a generic approach to formally verify autonomous vehicles with nonlinear behavior against reach-avoid requirements. The approach provides a UPPAAL timed-automata model of vehicle behavior, and uses UPPAAL STRATEGO for verifying the model with user-programmed libraries of collision-avoidance algorithms. Our experiments show the approach's effectiveness in discovering bugs in a state-of-the-art version of a selected collision-avoidance algorithm, as well as in proving the absence of bugs in the algorithm's improved version.

**My contribution:** I was the primary driver of the paper, proved the theorems, built the models, conducted the experiments, and wrote most of the text. All the remaining authors provided feedback and contributed with valuable ideas and comments.

# Chapter 2

# Preliminaries

In this chapter, we overview the preliminaries needed for the rest of the dissertation. First, we introduce the concepts of formal verification and synthesis, with an emphasis on model checking and statistical model checking as the verification techniques used in this dissertation. Next, we introduce the modeling languages and tools used in this dissertation: timed automata and UPPAAL, hybrid automata and UPPAAL SMC, timed games and UPPAAL TIGA as well as UPPAAL STRATEGO. Third, we introduce two novel algorithms for path finding and collision avoidance that are used in this dissertation. Last, we briefly introduce reinforcement learning and a classic reinforcement learning algorithm called Q-learning.

## 2.1 Formal Verification and Synthesis

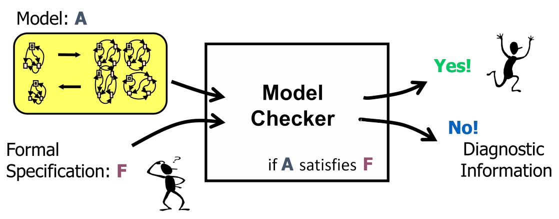
In this section, we briefly recall two techniques that serve as the main focus of this dissertation: formal verification and synthesis.

### 2.1.1 Formal Verification

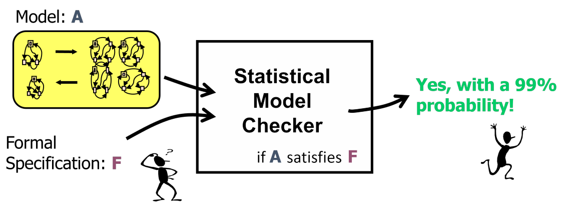
Formal verification is a process of mathematically checking whether a system, described by mathematical models, satisfies certain formal specifications, described in some logic (predicate logic, temporal logic, etc.) often called the property language [39]. Examples of mathematical models are finite-state automata [40] and Petri nets [41]. In this dissertation, we use model checking as the formal verification technique, and primarily focus on timed models, such as timed automata [19], for describing the agents' behaviors, because agents are real-time reactive systems. The properties that model checking is concerned with can be categorized into two classes:

- *Safety properties*: the model will never present a certain behavior, e.g., deadlock,
- *Liveness properties*: the model will present a certain behavior eventually, or infinitely often, e.g., a process waiting to enter a mutual-exclusive memory space will eventually enter.

In this dissertation, we particularly consider a subset of liveness properties regarded as timing properties, which require the models to not only eventually do something, but also do it within a time frame. This is justified by the fact that industrial systems are often concerned with timing properties for maintaining a certain level of productivity. For instance, an autonomous truck must transport 10 tons of materials within 30 minutes in a construction site.



(a) Exhaustive model checking



(b) Statistical model checking

Figure 2.1: Two formal verification techniques: exhaustive model checking and statistical model checking

As depicted in Fig. 2.1a, *exhaustive model checking* explores the entire state space of the model and returns a qualitative answer to the question of whether a model satisfies a formal specification (i.e., yes or no). When the answer is negative, the model checker can bring a counterexample (i.e., diagnostic information) that shows how the specification is violated by the model. Another verification technique called *statistical model checking* randomly simulates the model, samples its execution traces, and returns a quantitative answer to that question (e.g., the probability of a positive answer is 99%). A more detailed introduction of statistical model checking is given in Section 2.2.2.

### 2.1.2 Synthesis

Formal verification is concerned with proving that a system model satisfies a certain formal specification described by a set of logic-based properties. In contrast, synthesis is concerned with how to automatically construct a system model that provably satisfies a given set of properties [39]. In this dissertation, we formally prove the soundness of our synthesis methods, which means that the synthesis results are guaranteed to be correct (a.k.a., correct-by-construction). Additionally, synthesis can not only yield correctness-guaranteed results but also provide insights into inconsistent specifications when the answer of synthesis is negative.

In this dissertation, we propose two categories of synthesis methods: *search-based* methods, introduced in Sections 5.2.1 and 5.2.2, and *simulation-based* methods, introduced in Sections 5.2.2 and 5.2.3. The former methods are sound and complete, but not scalable, which means that the methods must find the valid result of synthesis when it exists (i.e., completeness) and the result must be correct (i.e., soundness), however, the methods cannot deal with complex models with large state space, e.g., when the number of agents is large. In contrast, the latter methods are scalable and sound, but not complete.

## 2.2 Modeling Languages and Tools

In this section, we introduce the formal modeling languages and tools that are used in this dissertation.

### 2.2.1 Timed Automata and UPPAAL

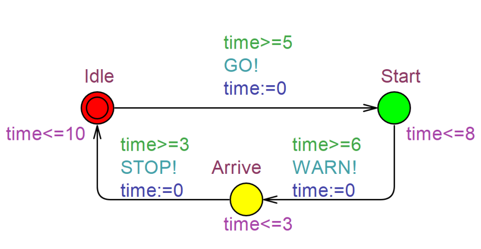
UPPAAL [20, 42] is a tool suite for modeling, simulation, and model checking of real-time systems. The modeling formalism of UPPAAL is UPPAAL *timed automata* (UTA), which is an extension of Alur's and Dill's *timed automata* that are finite automata extended with real-valued variables called clocks [19]. The latter models the logic clocks in systems, which are zero initially and then increase simultaneously at the same rate of one. UTA extends TA with data variables such as integers and Boolean variables. Formally, a UTA is defined as a tuple:

$$< L, l_0, A, V, C, E, I >, \tag{2.1}$$

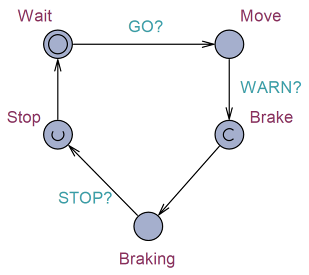
where $L$ is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $A = \Sigma \cup \tau$ is a set of *actions*, where $\Sigma$ is a finite set of *synchronizing actions* and $\tau \notin \Sigma$ denotes internal or empty actions without a synchronization, $V$ is a set of *data* variables, $C$ 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 $B(V)$, and $I : L \mapsto B_d(C)$ is a function assigning *invariants* to locations, where $B_d(C) \subseteq B(C)$ is the set of downward-closed clock constraints with $\bowtie \in \{<, \leq, =\}$. The invariants bound the time that can be spent at locations.

We illustrate the basics of UTA via an example. In Figure 2.2, one can see two UTA modeling a car and its controller, respectively. The automaton in Figure 2.2a models the controller, where *time* is a clock variable that measures the elapse of time and progresses continuously. The controller UTA has three locations, namely `Idle`, `Start` and `Arrive`, and edges connecting them. At each of the locations, the controller UTA has two non-deterministic choices of actions: (i) staying at the location and letting time elapse as long as the invariant on that location, e.g., `time <= 10`, is satisfied; (ii) moving via an edge to another location, as long as the guard on the outgoing edge, e.g., `time >= 5`, is satisfied. In Figure 2.2a, the controller UTA can stay at location `Idle` until time reaches ten, or transfer to location `Start` when time exceeds five.

A *network* of UTA (NUTA) models a parallel composition of UTA that can synchronize via *channels* (i.e., $a!$ is synchronized with $a?$ by handshake). When moving from location `Idle` to `Start`, the controller UTA synchronizes with the car UTA in Figure 2.2b, via a channel called



(a) UTA of a car's controller



(b) UTA of a car

Figure 2.2: An example of UTA in UPPAAL

`GO`. An exclamation mark "!" following the channel's name denotes the sender, and a question mark "?" denotes the receiver. Meanwhile, an assignment on the edge resets the clock, e.g., from `Idle` to `Start`, that is, the controller UTA sets the clock variable `time` to 0. The assignment can also be a function written in a subset of the C language, updating clock variables as well as data variables. In the UTA of a car (Figure 2.2b), there are two special locations, namely `Stop` and `Brake`. The former is called an *urgent* location and the latter is called a *committed* location. Both urgent and committed locations do not allow time to elapse, with the latter being stricter because the next transition is requested to start only from one of the committed locations of the NUTA.

The semantics of a UTA $\mathcal{A}$ is defined as a *timed transition system* over states $(l, v)$, where $l$ is a location and $v \in \mathbb{R}^C$ represents the valuation of the clocks in that location, with the initial state $s_0 = (l_0, v_0)$, where $v_0$ assigns all clocks in $C$ to zero. There are two kinds of transitions:

(i) *delay transitions*: $(l, v) \xrightarrow{d} (l, v \oplus d)$, where $v \oplus d$ is the result obtained by incrementing all clocks of the automaton with the delay amount $d$ such that $v \oplus d \models I(l)$, and

(ii) *discrete transitions*: $(l, v) \xrightarrow{a} (l', v')$, corresponding to traversing an edge $l \xrightarrow{g,a,r} l'$ for which the guard $g$ evaluates to *true* in the source state $(l, v)$, $a \in A$ is an action, $r$ is the clock reset set, and clock valuation $v'$ of the target state $(l', v')$ is obtained from $v$ by resetting all clocks in $r$ such that $v' \models I(l')$.

The UPPAAL model checker supports the verification of queries (properties) written in a decidable subset of Timed Computation Tree

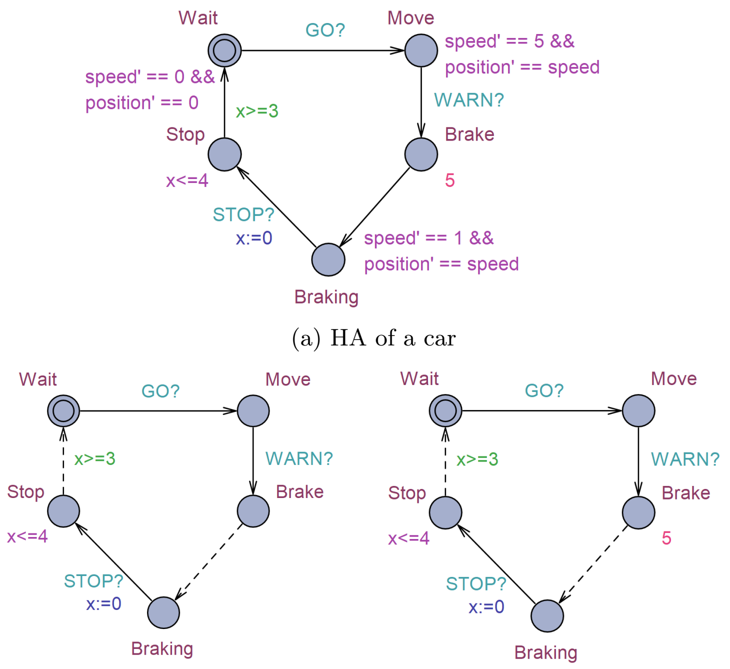
Logic (TCTL) [42]. The syntax of a TCTL formula consists of quantifiers over paths, and path-specific temporal operators. There are two types of path quantifiers: the universal one, "$A$" meaning "for all paths", and the existential one, "$E$" denoting "there exists a path". We are interested in two 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 dissertation 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 Reachability**: $E \Diamond_{\leq t}\ p$ means that there exists a path where $p$ is satisfied by at least one state of the path within $t$ time units.

### 2.2.2 Hybrid Automata and UPPAAL SMC

UPPAAL SMC [43] is an extension of the tool UPPAAL [43], which supports statistical model checking of *hybrid automata* (HA) [15]. HA in UPPAAL SMC are similar to UPPAAL TA, and are extended with a set of continuous variables whose derivatives are described by ordinary differential equations (ODE). Similarly to the UTA case, we illustrate the basics of HA via a simple example. Figure 2.3a shows the HA of a car with the same car controller as shown in Figure 2.2a. The derivatives of two continuous variables, namely *position* and *speed*, are defined in the HA. As variables *speed* and *position* represent the velocity and position of the car, respectively, based on Newtonian laws of motion, the derivative of *position* is the value of *speed* when the car is moving, and the derivative of *speed* is assumed to be five.

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 are refined based on probability distributions, either uniform distributions for time-bounded delays, or user-defined exponential distributions for unbounded delays. For example, in Figure 2.3a, the time-bounded delay at location `Stop` follows the uniform distribution, which means that it is equally likely to leave the location from time unit three to four, whereas the unbounded delay at



(a) HA of a car

(b) TG of a car

(c) STG of a car

Figure 2.3: Examples of HA, TG, and STG

location `Brake` follows an exponential distribution with an exponential rate 5. The exponential distributions make the HA become increasingly likely to leave the location as the delay increases at the location. When being executed, the HA race against each other according to their probability distributions, that is, they independently and stochastically decide how much to delay before making a discrete transition, with the "winner" being the automaton that chooses the minimum delay.

UPPAAL SMC supports an extension of *weighted metric temporal logic* [33] for probability estimation, whose queries are formulated as follows: `Pr[bound] (ap)`, where `bound` is the simulation time, `ap` is the formula based on either of the following two temporal operators: "*Eventually*" ($\Diamond$) and "*Always*" ($\Box$). Such queries estimate the probability that `ap` is satisfied within the simulation time bound.

### 2.2.3 Timed Games and Strategies

A *timed game* $\mathcal{G}$ (TG) [44] is a UTA whose actions $\Sigma$ are partitioned into controllable ($\Sigma_c$) and uncontrollable ($\Sigma_u$) actions. A TG is a useful mathematical model, suitable to describe a system consisting of several players that compete or collaborate to win a game, e.g., to finish their tasks. Note that each player can take arbitrary numbers of actions before the other player acts. The numbers depend on the design of the TG model. Informally, a strategy is a function that during the course of the TG constantly suggests the players what actions to do next. The suggestion is either a controllable action $a \in \Sigma_c$ or a delay. A winning strategy of a player is the one that always enables the player to win the game regardless of how others act.

When more information is known of the environment, for instance, the likelihood of uncontrollable actions or the probability distribution of delays, we consider *stochastic timed games* (STG) [45], where a stochastic environment is assumed. The environment makes choices of delay and uncontrollable actions stochastically, according to a density function for a given state. Correspondingly, strategies for STG are stochastic ones that indicate the probabilities of choosing controllable actions and delays at each state of the players.

Figures 2.3b and 2.3c depict a TG and an STG model of a car, respectively, with the same controller as in Figure 2.2a. The solid arrows are controllable actions and the dashed arrows are uncontrollable actions. Figure 2.2a and Figure 2.3b (respectively, Figure 2.3c) model a timed game (respectively, a stochastic timed game) between a car and its driving environment. The controller sends commands of moving and braking to the car via synchronization channels `GO`, `WARN`, and `STOP`, whereas the environment gets to choose the time of starting to brake and stop. In a timed game, the choices of uncontrollable actions are made non-deterministically, and the goal of a winning strategy can be to indicate the right actions to the controller at each of its states so that the car can fully stop eventually, after it starts to move. When the environment becomes stochastic, the model becomes an STG, where the choices of uncontrollable actions are stochastic. The winning strategy now involves controller actions in order to fully stop the car eventually, with the highest probability. The probability distributions in Figure 2.3c are the same as the ones in Figure 2.3a, that is, exponential distributions on unbounded delays (e.g., at location `Brake`) and uniform distributions on time-bounded delays (e.g., at location `Stop`).

#### UPPAAL TIGA and UPPAAL STRATEGO

UPPAAL TIGA [25] is an extension of UPPAAL, and it supports solving games based on TG with respect to the temporal properties mentioned in Section 2.2.1. In this dissertation, we use UPPAAL TIGA to solve our mission-planning problem as a solution of *search-based synthesis* of mission plans (details in Sections 5.2.1 and 5.2.2). UPPAAL STRATEGO [30] is a tool that integrates UPPAAL with two of its branches, that is, UPPAAL SMC [33] (statistical model checking) and UPPAAL TIGA [25]. In addition, it supports learning algorithms for solving STG, and we use this tool to develop our solution of *learning-based synthesis* of mission plans (details in Sections 5.2.2and 5.2.3).

## 2.3 Path Finding and Collision Avoidance

In this section, we briefly introduce a path-finding algorithm, namely Theta* algorithm, and a collision-avoidance based on dipole flow fields.

### 2.3.1 Theta* Algorithm

The Theta* algorithm has been firstly proposed by Nash et al. [23] to generate smooth paths with as few sharp turns as possible, from the starting position to the destination. Similar to the Dijkstra algorithm and A* algorithm [22], 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 dissertation, we use Manhattan distance [46] 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* [23], 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.

### 2.3.2 Collision Avoidance based on Dipole Flow Fields

When agents move, they may encounter moving obstacles and thus take a detour to avoid them. The problem is how far the detour should be from the planned path and how the agents can reach their destination when the deviation occurs.

Trinh *et al.* propose an approach to calculate *static flow fields* for all objects, and the *dynamic dipole fields* for the moving objects in the environment [38]. In the authors' method, every moving 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 moving object's velocity. As soon as the agents equipped with this algorithm get close enough to a moving obstacle, the magnetic moment around them keeps them away from each other. The static flow fields are created within the neighborhood of the initial path generated by the Theta* algorithm, and of the static obstacles. The force of the static flow fields is a combination of the attractive force drawing the agents to the initial path, and the repulsive force pushing the agents away from obstacles. Unlike the dipole field force, the static flow field force always exists, regardless of whether the object is moving or not. The combination of the static flow fields and the dynamic dipole fields enables agents to move safely by avoiding all kinds of obstacles and reach the destination, as long as a safe path exists. Compared with other methods [47, 48], this algorithm provides a novel method for path following and collision avoidance, in the shared working environment of humans and agents, which suits our requirements well. However, one needs to verify whether the algorithm is able to safely avoid all moving obstacles, including unforeseen ones. We carry out this verification as part of our dissertation contribution [18, 37] that is described later in Chapter 5.

## 2.4 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning aiming at calculating how agents should take actions in an environment, in order to maximize the accumulated reward obtained from the environment [49]. *Model-free RL*, such as *Actor-Critic algorithms* [50], relies on samples from the environment, which can be a model or a real environment, to estimate the rewards of the next state-action pairs. *Model-based RL*, such as *Dynamic Programming* [28], uses the model's predictions or distributions of the next state-action pairs and their rewards to calculate optimal actions. *Q-learning* is one of the model-free RL algorithms, which, at the limit, converges to *optimal policies* for reactive agents in a stochastic environment. Policies are associated with a state-action value function called *Q function*. The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \underset{a'}{\mathrm{MAX}}\, q^*(s',a')], \tag{2.2}$$

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 $s$, $\gamma$ is a discounting value, $s'$ is the new state coming from state $s$ by executing action $a$, and $\underset{a'}{\mathrm{MAX}}\, q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair $(s',a')$. The Bellman equation calculates the rewards of state-action pairs by considering both the current reward and the discounted maximum future reward. Watkins [51] shows that under the assumption of sufficient repeated sampling, the Q-learning algorithm converges towards the optimal Q-values and thus the solution to the Bellman equations. These values are stored in Q-tables, which serve as the strategies that we aim to synthesize.

# Chapter 3

# Research Problem

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

## 3.1 Problem Description

The broad focus of this dissertation is on *scalable mission plan synthesis of multi-agent systems (MAS)*, and *reach-avoid verification* of such systems. Concretely, we target two aspects, that is:

1. synthesizing mission plans of MAS that provably satisfy certain requirements;
2. ensuring the reach-avoid requirement when agents are executing the mission plan and face static and dynamic obstacles.

The first aspect focuses on computing plans that guide the agents towards their destinations, respectively, and ensuring that agents execute various tasks in compliance with certain rules, such as a certain execution order and timing requirements. The second aspect needs to consider the kinematics of the agents and moving obstacles. Therefore, when synthesizing mission plans, one only needs to consider the discrete feature of the environment and the agents. More specifically, only static obstacles and important positions, a.k.a. milestones, where tasks are carried out, are extracted from the environment and modeled. The continuous movement of the agents as well as the operations for executing the tasks do not need to be modeled when our target is only about synthesizing mission plans. Therefore, agent movements and task executions can be modeled as time intervals between the best-case execution time (BCET) and the worst-case execution time (WCET). Although a continuous concept, time, is included in the models, by certain techniques of abstraction, such as zones [52], the state space of the models can be represented symbolically, and the numbers of such symbolic states are countable. Hence, our **problem of mission plan synthesis** is to somehow traverse the symbolic state space of a model of MAS and find the traces (sequences of states connected by transitions) that satisfy our requirements. As a variant of the classic *job-shop scheduling* problem, mission plan synthesis is NP-hard [53], that is, when the number of agents grows linearly, the state space of the model increases exponentially, making the traversal of the state space impossible to finish within a reasonable time when the agents are many. In summary, solving mission planning is connected to answering the following research questions:

*How to find the paths that autonomous agents need to follow towards their destinations, respectively, and avoid static obstacles, and how to schedule the actions of movement and task execution so that the agents finish all tasks timely, while obeying the task constraints?*

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 **dynamic reach-avoid requirement**, where we take into account the continuous behaviors of agents and unforeseen moving obstacles. When the agents move and execute the synthesized mission plan, they must be ensured to follow the planned paths closely for the collision avoidance against the known and static obstacles, deviate from the planned paths timely for the collision avoidance against the unforeseen moving obstacles, and reach their destination eventually. How comprehensively and faithfully the model reflects the real scenarios is one important factor in this subproblem, while the ability of verification must still be preserved by the solution as we focus on generating a correctness- and safety-guaranteed design of agents. In summary, solving the reach-avoid verification is connected to answering the following research questions:

*When no moving obstacle is around agents, how to ensure the agents to follow their planned paths closely enough so that no collision occurs?*

*When moving obstacles appear, how to ensure the agents to deviate from their planned paths timely so that no collision occurs?*

*When deviations from the agents' planned paths occur, how to ensure the agents to reach their destinations eventually, respectively?*

The overall research problem is the combination of the research problems connected to the two mentioned aspects and is presented as follows:

*How to synthesize correctness-guaranteed mission plans for multi-agent systems, and verify their reach-avoid functionalities when the agents are executing the mission plans, so that the agents are functionally correct and safety-guaranteed, with the scalability tamed as the number of agents increases?*

## 3.2 Research Goals

To solve the research problem as formulated in Section 3.1, the main research goal of the dissertation is as follows:

**Overall goal.** Determine how algorithmic formal methods can be employed and scaled up to synthesize and verify autonomous systems with respect to mission planning and dynamic collision avoidance.

**Remark 1.** *The algorithmic formal methods, e.g., model checking, are based on searching the state space of formal models, and are used for mission plan synthesis and reach-avoid verification. We do not consider deductive methods of formal verification in this dissertation.*

A key principle of designing complex systems in general is the notion of separation of concerns [54]. The essential idea is to avoid co-locating different concerns within the design. In our research problem, mission planning is not concerned with the agents' operations in a continuous dynamic environment, whereas the continuous behavior of the agents is for executing the pre-computed mission plans. Therefore, achieving the separation of concerns of mission planning and continuous movement and task execution, while backing them by formal verification techniques, can be beneficial. Consequently, the first subgoal is as follows:

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

The context of mission plans varies, for example, collaborative environments give agents the complete control over their actions, whereas stochastic and non-deterministic ones only let the agents decide when to start their actions but choose the ending times independently. We need to adopt suitable techniques for different kinds of problems and identify their advantages and limitations. This gives rise to the second subgoal, formulated as follows:

**Subgoal 2.** Assuming different types of environments of mission planning, that is, deterministic and collaborative ones, stochastic ones, or non-deterministic ones, identify suitable methods for mission planning, and evaluate their appropriateness in various scenarios.

As mission planning for multiple agents is an NP-hard problem [53], the complexity of the problem can grow dramatically with the linear increase of agent numbers. Consequently, the synthesis of mission plans can be computationally intensive and the resulting mission plans can take large memory space that is a disadvantage in most cyber-physical systems, such as MAS. Hence, the third subgoal aims to provide a scalable method for mission plan synthesis and compression. The subgoal is formulated as follows:

**Subgoal 3.** Provide scalable approaches that involve model checking, for mission-plan synthesis and compression of MAS, such that the computation time is reasonable when the number of agents is large and the resulting mission plans only contain the useful information that guarantees the agents to satisfy given temporal requirements.

**Remark 2.** *In this dissertation, the number of agents is considered large when it is more than four, which aligns with the context of certain industrial systems, such as autonomous quarries [21].*

After addressing Subgoals 1-3, when agents start to execute the mission plans, can they follow the reference paths closely enough so that collisions with known static obstacles can be **avoided**? More so, when dynamic obstacles appear, can the agents stop or turn timely so that collisions are avoided? When facing unforeseen obstacles, both static and dynamic, can the agents be guaranteed to **reach** their destinations in a timely fashion? The fourth subgoal of our research aims to answer these questions, by focusing on the continuous behavior of the agents and dynamic obstacles:

**Subgoal 4.** Ensure the reach-avoid requirement of agents when executing mission plans in environments with unforeseen static and dynamic obstacles.

Our solutions need to be aligned with the state-of-the-art and state-of-practice, in the sense that the approaches are theoretically innovative and practically applicable. Hence, our fifth subgoal is:

**Subgoal 5.** Develop automated support that integrates the approaches of mission plan synthesis and verification, and assesses the applicability of the approaches in 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 dissertation. If achieved, subgoal 1 brings an overview of the research problem and decouples it into suitable modules such that the inherently different components of the problem can be solved separately. Subgoal 2 focuses on the subproblem of mission planning. It enables us to select the right tools and motivates us to invent new methods to deal with different environments, which sets the foundation of addressing the problem in the next subgoal. Subgoal 3 concentrates on providing model-checking-based concrete solutions for mission plan synthesis and compression that are suitable for different kinds of environment. Once mission plans are synthesized, Subgoal 4 continues towards ensuring the correctness of agents executing the mission plans in the sense of fulfilling the reach-avoid requirements. Finally, if the tool support in Subgoal 5 is realized, all the approaches can be integrated, which facilitates their applicability and enables experimental evaluations on industrial systems. 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 research 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 dissertation.

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. Based on the indus-

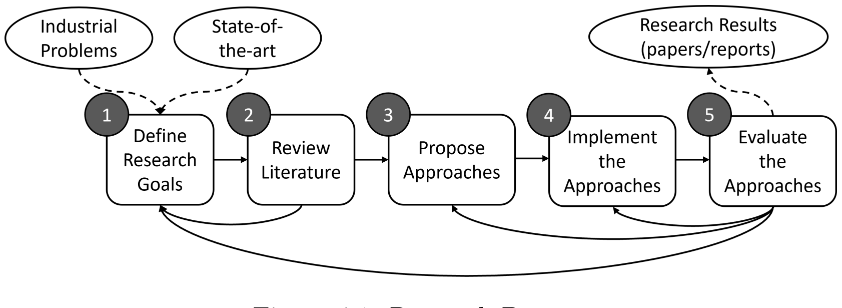


Figure 4.1: Research Process

trial problems and the state-of-the-art research results, we formulate the research goals (Step 1). To address these goals, we start by investigating the formal methods and their applications on agents that have been studied by industry and academia. The research method that we adopt in this phase is *critical analysis of the literature and practice* method [55].

By reviewing literature (Step 2), we understand the state-of-the-art and state-of-practice related to the topic that we aim to research, and identify the gaps between the problems and the existing solutions so that we can modify the research goals if necessary. The next action is to propose approaches to solve the problems, and implement the approaches within a prototype tool that could be applied to industrial applications (Steps 3 and 4). Finally, we apply the approaches and the tool to an industrial case study to demonstrate their applicability to real-world systems, and conduct several groups of experiments assuming different environments to evaluate their performance. The results comprise a series of research papers and reports (Step 5). Whenever the results match the expectation of a research goal, we conclude that the goal is achieved. Whenever the results differ from our expectation, either for better or worse, we analyze the reasons behind the deviation. Then, if necessary, we use the obtained experience to modify the research goals, or propose a new and improved solution to the same problem (Step 5 to Step 1, or Step 3, or Step 4).

We have applied a set of research methods during the activities of the research process. During the design and evaluation of our approach for synthesizing mission plans and pattern-based modeling (details are in Section 5.3.1), we apply the *proof of concepts* method [56] to show the correctness and applicability of our proposed approach. In *Paper B* and *Paper E*, we integrate our methods for strategy synthesis and compression into the tool UPPAAL STRATEGO. In *Paper C*, we present our tool *MALTA* that provides a GUI for environment and mission configuration, and extensible modules for path finding and task scheduling. The tools are the proof of concepts of our model-checking-based mission-planing methods in different environments. When validating the research results in the industrial use case, we employ the *proof by demonstration* method [56], by applying our tool *MALTA* assuming an industrial-like context. In *Paper B*, *Paper C*, *Paper E*, and *Paper F*, we experiment different scenarios of the industrial use case to demonstrate the tool's ability of solving practical problems. We also use *mathematical modeling* and *mathematical proof* [56] in showing the correctness of our synthesis methods (*Paper B*) and model transformation from the continuous trajectories to discrete-time trajectories of agents (*Paper F*).

# Chapter 5

# Dissertation Contributions

In this chapter, we present the contributions of this dissertation, which address the aforementioned research goals. We first introduce a two-layer framework for modeling and verifying agents, which facilitates the design of our methodology, to address the overall research goal. Next, we present each of the techniques and algorithms that support the proposed methodology, and address the subgoals, respectively. We also indicate which included paper contains a particular contribution, as well as the research subgoal that the latter addresses.

## 5.1 A Two-Layer Framework for Modeling and Verifying Autonomous Systems

In this section, we briefly introduce the two-layer framework for agent modeling and verification, which is initially proposed in **Paper A** [18]. We first give the overall description of the framework, before introducing the communication between the layers and the probabilistic mission planning based on this communication, which are in **Paper D** [32].

### 5.1.1 Overall Description of the Framework

Two of the fundamental functionalities of agents are *static mission planning* and *dynamic collision avoidance*. The former includes *path finding* and *task scheduling*, which we call *mission planning* collectively. When synthesizing a mission plan, the autonomous systems (agents) focus on generating paths and an order of executing tasks. The generated mission plan must satisfy various requirements. For instance, in an autonomous quarry, autonomous trucks should carry 100 $m^3$ of stones to the primary crusher that outputs crushed stones at given fractions, before carrying the crushed stones to the secondary crusher, all within 2 hours. At this level of design, the agent model should not include the concrete movement and operation, because they are irrelevant in mission plans. Consequently, the agent model only needs to know what to do or where to go at certain time points and positions in the environment.

At the lower level, when the agents start to execute a mission plan, their continuous behaviors have to be modeled, such that the models are as realistic as possible. Hence, it is better to model behaviors like acceleration and turning at this separate level. Therefore, mission planing and the continuous behaviors can be decoupled to reduce the complexity of the design and verification, and provide a separation of concerns.

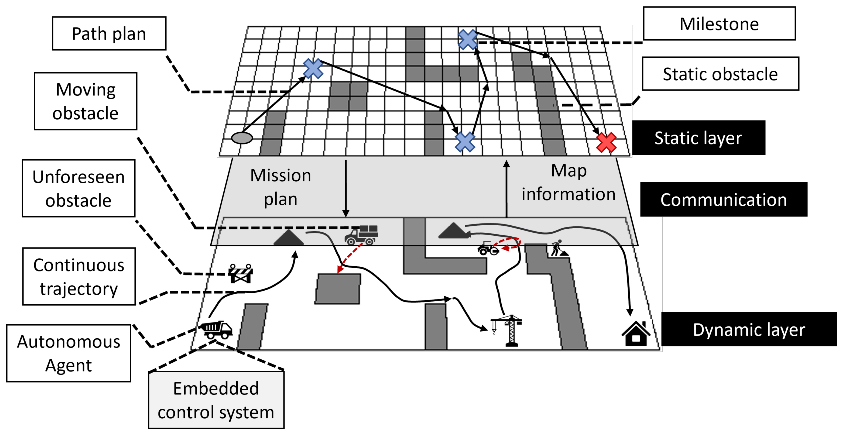
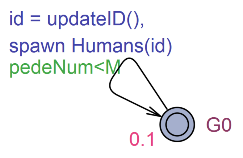
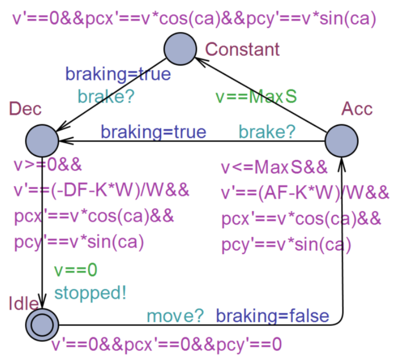


Figure 5.1: A two-layer framework for formally modeling and verifying multi-agent autonomous systems

In **Paper A** [18], we propose a two-layer framework consisting of a *static layer* and a *dynamic layer*, which is depicted in Fig. 5.1. The *static layer* is responsible for mission planning, based on the known information of the environment, and includes static obstacles and mile-



(a) A HA generating pedestrians



(b) A HA modeling the movement of agents

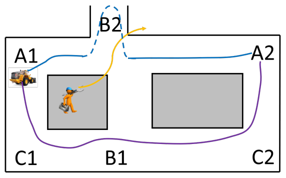
Figure 5.2: Examples of HA in the dynamic layer of the framework

stones where the tasks should be carried out. Moving obstacles that are unforeseen by the agents are considered in the *dynamic layer*, which is designed to simulate and verify the agents to guarantee that they follow the reference path generated by the *static layer* and avoid the moving obstacles dynamically. These two layers support the modeling of the discrete behavior and the continuous behavior separately, such that the desired decoupling is achieved.

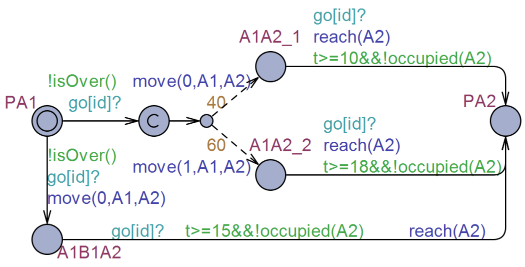
### 5.1.2 Communication between the Two Layers

The communication between these two layers is needed in order to enable the exchange of information. For example, when agents deviate too far from the planned paths, they can send the new map of the environment including the newly discovered static obstacles to the static layer. The latter can synthesize a new mission plan and send it to the dynamic layer so that the agents can adjust their behavior according to the new plan and verify it. The newly discovered obstacles can appear stochastically, which requires the static layer to synthesize mission plans based on the probabilistic information of the environment. **Paper D** [32] demonstrates the communication between these two layers and shows the ability to re-plan in the presence of unexpected pedestrians.

Next, we introduce the models used in both layers and how the communication works. As depicted in Fig. 5.3a, in an environment with an



(a) An intersection containing pedestrians



(b) A stochastic model of agent movement

Figure 5.3: A scenario of intersection and a stochastic model used in the static layer for mission planning

autonomous wheel loader and some pedestrians that can possibly cross the intersection near *B2*, we need to find the optimal path for the wheel loader to travel safely and efficiently from *A1* to *A2*. First, we simulate the stochastic occurrence of pedestrians by using the stochastic hybrid automata (SHA) that are initially proposed in **Paper A** [18] and used in the *dynamic layer*. Fig. 5.2a is an example of such a SHA in UPPAAL SMC, which stochastically transfers via the self-loop edge of location `GO`. The transition follows an exponential distribution with a rate of 0.1. One can modify the rate to change the possibility of the occurrence of the pedestrians. The function `spawn Humans(id)` generates instances of this SHA template during the verification time to mimic the stochastic occurrence of pedestrians.

When the wheel loader encounters a pedestrian, it may stop, or accelerate, or take a detour via *B2*. The choices depend on the concrete collision-avoidance algorithm. In our study, we employ a novel algorithm based on dipole flow fields [38]. The algorithm enables the agents to deviate from their planned paths in order to avoid dynamic obstacles. The detour can prolong the traveling time that is even longer than the traveling time on another road, from *C1* and *C2* and finally to *A2* (see Fig. 5.3a). The length of the prolonged time depends on how many pedestrians exist in the intersection and their moving speeds, which are stochastic.

To obtain the lengths and probabilities of the prolonged traveling time, in **Paper D** [32], we reuse the HA model of agents that is initially designed in **Paper A** [18] (see Fig. 5.2b). Next, we model-check the composed HA model in Fig. 5.2 against a reachability property in the form of Query (5.1) supported by UPPAAL SMC:

$$\texttt{Pr[<=T](}\Diamond\ \texttt{A2),} \tag{5.1}$$

where `T` is the simulation time, `A2` is a Boolean variable indicating whether the wheel loader arrives at position *A2* or not. UPPAAL SMC also supports verifying properties in the following form:

$$\texttt{Pr[<=T] (}\Box\ \texttt{A2 imply t<=PT),} \tag{5.2}$$

where `t` is a clock variable for measuring the traveling time from *A1* to *A2*, and `PT` is a constant value for estimating the traveling time. Query (5.1) checks whether the wheel loader ever gets to the position *A2*. Query (5.2) checks if the traveling time is at most `PT` time units when the wheel loader arrives at *A2*. By changing the value of `PT`, we can obtain the probabilities of reaching *A2* within different time. The estimation is reflected in the model in Fig. 5.3b, which is used in the *static layer*. Here, we can see that traveling from *A1* to *A2* via *B2* takes 10 time units for 40% of probability, and 18 time units for 60% of probability, whereas traveling via *B1* takes a fixed traveling time: 15 time units. Mission-planning algorithms based on this stochastic model, which are introduced in Section 5.2, can generate the statistically optimal strategy that most likely takes an agent to its destination the fastest. In this contribution (**Paper A** [18] and **Paper D** [32]), which addresses **Subgoal 1**, we show the two-layer framework, its ability of decoupling the problems of mission planning and dynamic collision avoidance, and the communication between the two layers.

## 5.2 Mission Planning in Different Environments

*Problem description.* Mission planning for multi-agent autonomous systems (MAS) is separated into two subproblems: path finding and task scheduling. The former cares about finding a static path that avoids static obstacles and reaches the destination. The latter aims at scheduling the tasks of agents, e.g., digging and loading in quarries, such that the agents can finish their tasks in time with respect to temporal constraints. Although they are different, these two subproblems are essentially about ordering the motion primitives (a.k.a. actions) of agents, such as moving to a milestone where a corresponding task can be carried out, or starting to execute a task at the current milestone.

Each of the agents in the MAS is assigned with a group of tasks. To achieve the global goal that the MAS is designed to accomplish, each of the agents must carry out tasks individually if the corresponding tasks are independent, or collaboratively if the corresponding tasks need agents to execute at the same milestone during the same time period. In either case, the requirements, e.g., achieving the global goal within a certain time frame, must be met. For instance, in a quarry, the global goal is to dig and transport a certain amount of stones in 24 hours. To achieve this global goal, autonomous wheel loaders need to dig stones at stone piles, and collaborate with autonomous trucks to load the stones into the trucks. The latter then proceed to transport the stones to each of the crushers and unload stones. When a special event happens, e.g., fuel level becomes low, the corresponding agent needs to react to it in priority, e.g., moving to a charging pole. Mission planning in this quarry is to order the autonomous wheel loaders' and autonomous trucks' actions of moving and executing a task, so that they can travel without collision and achieve the global goal by carrying out each of their tasks.

One of the challenge is the variable execution time of actions. For example, when the autonomous trucks load stones into crushers, the speed of the conveyor belt of a crusher can vary in a certain range, which influences the execution time of loading stones, and further influences the entire mission plan. Section 5.2.2 introduces this challenge in details. The uncertainty of action time is caused by the factors in the environment, which is not controllable by the agents. Therefore, assuming different types of environments, the solution for mission planning can be different. In summary, in the context of MAS, mission planning is formulated as follows:

*Mission planning*: Given a MAS and a set of requirements, the goal of mission planning is to order the agents' actions of movement and task execution, according to their variable ending time and occurrences of events, which are decided by the environment, such that the MAS can finish its tasks and satisfy the requirements.

**Remark 3.** *One can reduce the planning problem to a path-finding problem by removing the actions of task execution, or a task-scheduling problem by removing the actions of movement. Our algorithms must be capable of solving the general problem that contains path finding and task*

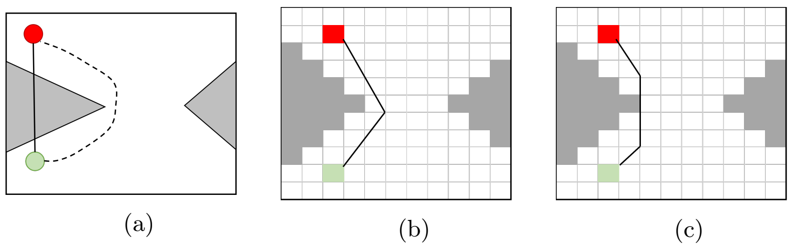

(a) (b) (c)

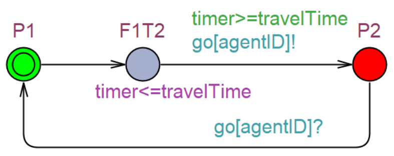
Figure 5.4: Examples of trajectories and the granularity of movement.

*scheduling or only one of them.*

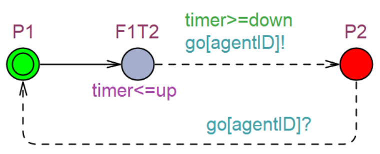
**Remark 4.** *The requirements can be functional ones, such as "always start task A after task B finishes", and safety ones, such as "no collision with the static obstacles in the environment happens".*

The agent actions are modeled into different granularity according to different purposes of mission planning. For example, if we reduce the planning problem to a path-finding problem, as depicted in Fig. 5.4, the movement from the red dot to the green dot can be modeled as one step in Fig. 5.4a or several steps in Figures 5.4b and 5.4c. Different granularity serves different purposes of the path plans. Path plans based on maps like Fig. 5.4a can serve as a high-level plan (the solid line) that indicates which milestone to visit and when. A low-level algorithm for collision avoidance is needed in this case, which must enable the agents to avoid all kinds of obstacles (the dashed line). Path plans in maps like Figures 5.4b and 5.4c show the concrete steps of movement to avoid the static obstacle between the initial and ending positions. Note that the collision avoidance against moving obstacles and other agents is considered in the dynamic layer of our two-layer framework [18], which is not a concern of mission planning.

*Modelling agent movement.* No matter which granularity is chosen, the agent movement can be modeled as the ones in Fig. 5.5, where locations `P1` and `P2` represent the current position and the target position of the movement, respectively, location `F1T2` models the duration of traveling. Fig. 5.5a models the movement in a collaborative environment, where the traveling time of each agent is fixed, since any agent can fully control the starting and ending time of its actions. Fig. 5.5b models the movement in a non-deterministic environment, where an agent only



(a) A UTA of agent movement



(b) A TG of agent movement

Figure 5.5: Models of agent movement in different kinds of environments

decides the starting time of its actions but the ending time is decided by the environment non-deterministically within a time interval, which is between the constant integers `up` and `down` in the Figure. When the environment is stochastic, for example, the probabilities of reaching `P2` between `up` and `down` follow an uniform distribution, the TG models can be interpreted as a STG with no change on the model template. Similarly, task executions can be modeled as UTA or TG models depending on the type of the environment. Detailed description of the models can be found in **Paper B** [26].

*An overview of the mission-planning solutions.* Given the agent models, mission planning is about exploring the state space of the models and finding the traces that solve the planning problem of MAS. We categorize the planning problem into three categories (Table 5.1).

- When the environment is assumed to be collaborative, that is, an environment whose behavior is fully controlled by agents, the problem of mission planning is a 1-player game. For this kind of environments, we propose a method based on UTA and model checking in UPPAAL named *TAMAA (Timed Automata based Mission planning for Autonomous Agents)* (**Paper B** [26]).

- When the environment is non-deterministic, the problem becomes a 2-player game, and we can use UPPAAL TIGA [25] or our method *MCRL (Model Checking + Reinforcement Learning)* to solve it [29]. MCRL is a novel approach that combines model checking with reinforcement learning, which is capable of dealing with a large number of agents that TAMAA cannot deal with (**Paper B** [26]).

- When the environment is stochastic, the problem becomes a $1\frac{1}{2}$-player game and can be solved by UPPAAL STRATEGO that includes UPPAAL SMC [30] (**Paper D** [32]).

Technical details of the above mentioned approaches are introduced in Sections 5.2.1, 5.2.2, and 5.2.3. Their strengths and weaknesses are presented in **Paper B** [26].

Table 5.1: Summary of approaches for different games

| | TAMAA | TIGA | MCRL | STRATEGO |
|---|---|---|---|---|
| Model | UTA | TG | TG & STG | STG |
| Game | 1 player | 2 player | 2 player | $1\frac{1}{2}$ player |
| Techniques | Model Checking [13] | Symbolic On-The-Fly Algorithm [44] | Reinforcement Learning & Model Checking [29] | Reinforcement Learning [30][45] |

In summary, in these contributions, mission planning is uniformly defined as a planning problem of MAS, based on which the solutions for solving the problem in different types of environments are categorized and proposed. This contribution addresses **Subgoal 2**. In the next subsections, we give a more detailed description of our proposed mission-planning methods.

### 5.2.1 Mission Planning in Deterministic Environments

In the previous section, we have categorized mission planning in deterministic environments as a 1-player game, in which the goal of winning the game is to order the agents' actions so that they can finish their tasks and reach their destination in the shortest time. TAMAA is the method that we propose for solving 1-player games, which provides an automation of model generation for constructing the UTA of agent movement and task execution [13]. Figure 5.6 depicts the workflow of

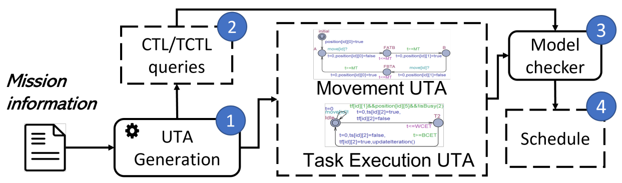
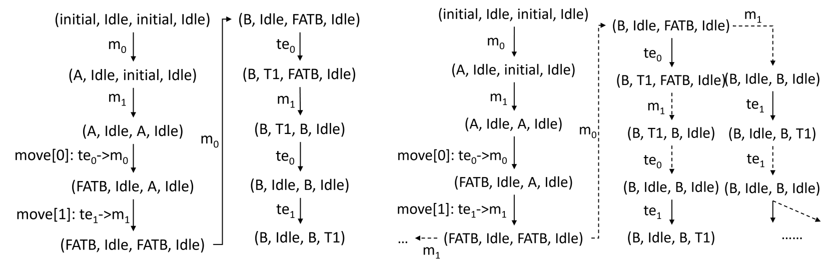


Figure 5.6: Workflow of TAMAA

TAMAA, where the mission information refers to the topology of the environment and task constraints, etc., and the UTA generation module generates the movement and task execution UTA automatically. For example, Fig. 5.5a is the movement TA that is generated by TAMAA. The automatically generated models are also used in other mission-planning methods (Table 5.1) with slight changes, which are introduced in detail in Subsections 5.2.2 and 5.2.3.

When the models are generated, the mission planning algorithms need to somehow explore the state space of the models and find out the execution traces that satisfy the requirements, e.g., finishing the tasks eventually, within one hour. TAMAA uses UPPAAL [20] to verify a generated UTA model and obtain the fastest execution trace in case the model satisfies given requirements. Here, we only show the TCTL property of the *Timing* requirement used in UPPAAL. The rest of the properties are reported in our work [13] and **Paper B** [26] in this dissertation. The TCTL reachability Query (5.3) checks if agents can repetitively execute their tasks `M` rounds within `TL` time units, where `ite` is an integer array storing each agent's iteration of its tasks, that is, finishing all tasks once counts for one round, `x` is a clock variable that is never reset, `N` and `M` are two integers indicating the number of agents and the requested iterations of tasks, respectively:

$$\texttt{E}\Diamond\ \texttt{((forall(i:int[0,N-1]) ite[i]>=M) \&\& x}\leq\texttt{TL)} \qquad (5.3)$$



(a) An example of a trace    (b) An example of executing a strategy

Figure 5.7: Examples of: (a) UTA trace in TAMAA, and (b) executing a TG strategy in UPPAAL TIGA

In case the Query (5.3) is satisfied, UPPAAL returns a trace that reaches the goal state within the shortest time, or the fewest steps, or in an arbitrary manner. In TAMAA, we always use the fastest trace to get the order of actions: movement or task execution. Fig. 5.7a depicts a segment of such traces belonging to a model of 2 agents, where states of the models are symbolically represented by the locations of the UTA, e.g., `(A, Idle, initial, Idle)` represents a state where the movement UTA of an agent is at location `A`, and its task execution UTA is at location `Idle`, whereas another agent's movement UTA and task execution UTA are at locations `Initial` and `Idle`, respectively. The symbols $\texttt{m}_i$ and $\texttt{te}_i$ stand for actions in the movement and task execution UTA of agent $i$, respectively, and $\texttt{move[i]:te}_i\texttt{->m}_i$ stands for the synchronized actions of starting to move. As depicted, all the actions are controllable by the agents (i.e., solid lines), which consecutively or alternately move the respective agent and execute tasks. Two or more agents can stay at the same milestone (e.g., `B`) but cannot execute the same task (e.g., two `T1` cannot appear at the same state) unless the agents are not mutually exclusive in the task.

Finally, when such a trace is found, a mission plan is generated based on the trace, which contains the order of actions and the paths towards the destination that avoid static obstacles. Since the resulting mission plans are based on the fastest traces that satisfy given requirements, they are guaranteed to meet the requirements too, and complete the tasks in the shortest time. Additionally, as the search of the model state space is exhaustive in UPPAAL, TAMAA can eventually find such a trace to synthesize a mission plan. Formally, if a 1-player game of mission planning has a winning solution, TAMAA is able to find it; when TAMAA finds a winning solution of a 1-player game of mission planning, it is guaranteed to be correct.

Next, we introduce the solutions for solving 2-player games where environment becomes non-deterministic, and $1\frac{1}{2}$-player games where environment becomes stochastic.

### 5.2.2 Mission Planning in Non-deterministic Environments

**TAMAA in UPPAAL TIGA: non-deterministic environments.** When the environment becomes non-deterministic, agents only choose which actions to start and when. The ending time of actions is decided by the environment, non-deterministically. Therefore, the mission-planning problem now becomes a 2-player game, where agents and the environment represent the players, respectively. The goal of winning the game

Table 5.2: Performance evaluation of synthesis in UPPAAL TIGA with different number of agents running 3 tasks among 3 milestones

| Number of agents | Number of explored states | Computation time |
|---|---|---|
| 2 | 775 | 5 ms |
| 3 | 222,88 | 220 ms |
| 4 | 764,001 | 18.1 s |
| 5 | 33,312,229 | 53.8 mins |
| 6 | Out of memory | Unknown |

now is to find out a strategy that controls the agents' controllable actions in order to finish their tasks while satisfying given requirements, no matter when and which uncontrollable actions are taken by the environment.

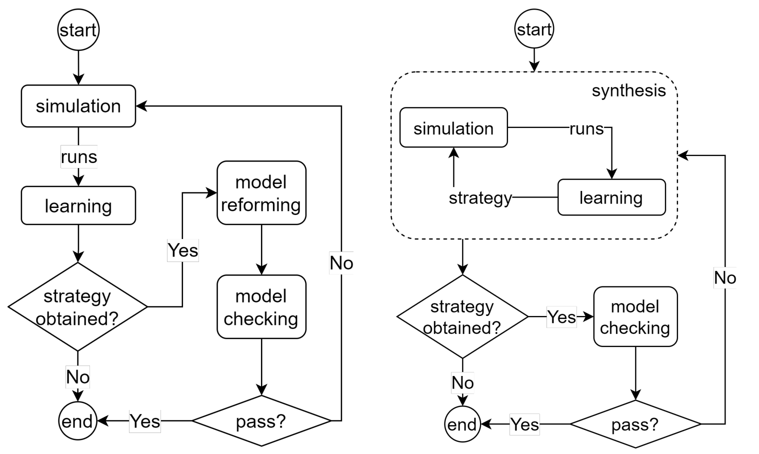
As depicted in Fig. 5.5b, the agent actions are modeled as *timed games* (TG) [44] in UPPAAL TIGA. UPPAAL TIGA is a tool for modeling and synthesizing strategies for TG. It uses a symbolic on-the-fly algorithm to explore the state space of a TG model, and finds the controllable state-action pairs that win a game. Fig. 5.7b shows an example of running a strategy of TG in UPPAAL TIGA, which contains controllable (solid lines) and uncontrollable actions (dashed lines). The first four steps in Fig. 5.7a and Fig. 5.7b are the same, being all controllable actions. From the fifth step in Fig. 5.7b, the strategy starts to be different from the trace, because the former has uncontrollable actions, which means that the environment decides which actions to perform instead of the agents.

The same as TAMAA in UPPAAL for 1-player games, TAMAA in UPPAAL TIGA is based on exhaustive graph search, and thus it is *sound*, that is, the resulting mission plans are guaranteed to be correct, and *complete*, that is, if a correct mission plan exists in the model, the method will terminate and find this mission plan. In addition, the result of synthesis in UPPAAL TIGA is a set of permissive strategies that are not necessarily optimal but guaranteed to satisfy the requirement anyway. Although UPPAAL TIGA compensates TAMAA in UPPAAL by solving the 2-player games, we discover that both methods are not able to solve the mission-planning problem with more than 5 agents [13, 26]. Table 5.2 shows the performance of running TAMAA in UPPAAL TIGA for solving the problem of agents running 3 tasks in an environment with 3 milestones. The computation time increases dramatically from 4 agents to 5 agents. When the agents are 6, the tool runs out of the physical memory that is assigned by the operating system (i.e., 2 GB on Windows 10). Therefore, we need to develop a scalable method to deal with a large number of agents.

In **Paper B** [26], we analyze the root of the state-space explosion problem in our mission plan synthesis. The problem is caused by the exponentially increased number of states when multiple agent models are composed together. The interleaving behavior of the agents makes the composed model extremely complex to be explored and analyzed. For example, when agents A, B, and C are traveling to a common and mutually exclusive position, who arrives first yields different situations. Moreover, the traveling time of the agents are intervals, which complicates the problem even more. The graph-search based methods [13, 44] rely on the algorithmic exploration of the state space, which traverses the states in a fixed order (e.g., the depth-first order), and needs to store all the explored states. Even though the symbolic on-the-fly algorithm of UPPAAL TIGA alleviates the state-space explosion to some extent, the nature of the algorithm still limits the size of its solvable problems.

**MCRL: scalable synthesis**. In an attempt to improve scalability of mission-plan synthesis, we propose a novel approach of synthesis, namely *MCRL (Model Checking + Reinforcement Learning)*, which replaces the exhaustive graph search with *random simulation and reinforcement learning* [29]. The approach employs Monte-Carlo simulation to explore the model state space, uses a reinforcement learning algorithm, i.e., Q-learning [51], to generate a strategy, and verifies the strategy by model checking to ensure the correctness of the synthesized strategy. The Monte-Carlo simulation solves the out-of-memory error because it randomly explores the model state space, which means it does not need to store all of states but only samples of traces, and the exploration terminates when the simulation time ends.

Fig. 5.8a depicts the first version of MCRL, where we use Q-learning to process the traces and compute the scores of state-action pairs by using Equation (2.2) (shown in Section 2) and then populate a Q-table to store the scores. The Q-table serves as the strategy (mission plan) that we aim to synthesize. Next, we need to check if the strategy indeed satisfies the requirement. Before running model checking, we reform the agent model by adding a new UTA called *conductor*, which is responsible for searching the strategy and returning the action having the highest score to the agent model at each of its states. Therefore, when the



(a) MCRL 1.0 [29] (b) MCRL 2.0 [26]

Figure 5.8: Workflow of MCRL

new model of the MAS is checked by UPPAAL, the agents are controlled by the conductor model based on the information in the strategy. If the verification passes, then the strategy is guaranteed to be correct in the sense that the new agent model can satisfy the requirements when it is under the control of the strategy. If the verification fails, a loop going back to simulation and learning takes place with the number of required sampling traces being increased so that the learning can be more thorough than the previous round of synthesis.

At the step of model checking (Fig. 5.8a), one typical requirement is a liveness property of the model, captured by Query (5.4), where $\phi$ is the goal that the agents need to satisfy eventually, e.g., finishing all required tasks. Query (5.4) means that the agents must always eventually satisfy $\phi$ no matter how the environment acts.

$$\mathtt{A}\Diamond\ \phi \tag{5.4}$$

The first version of MCRL is realized in UPPAAL, where a step of model reforming must be carried out before model checking, which complicates the modeling phase [29]. Moreover, the simulation and learning are separated in two steps so they cannot benefit from each other.

In **Paper B** [26] and **Paper E** [31], we collaborate with researchers from Aalborg University in Denmark to integrate MCRL into the tool UPPAAL STRATEGO. The integration enables MCRL to conduct the simulation and learning in an interleaving manner. As depicted in Fig. 5.8b, at the step of synthesis, simulation and learning iterates until a user-configured number of traces is sampled and finally produces a strategy. An intermediate strategy that may not be complete is used in the simulation. The temporary information of the scores of state-action pairs reflects the experience accumulated in learning so far, which can benefit the simulation to reach the goal states more likely than totally random simulation. Specifically, actions with higher scores become more likely to be chosen by the simulator. Unexplored actions enjoy the same priority to be chosen as the ones with the highest scores. Eventually, when a strategy is produced, it is model checked by the model checker in UPPAAL STRATEGO, without the extra work of constructing the *conductor* UTA, because we extend the model checker by leveraging its new function of calling external libraries (**Paper B** [26] and **Paper E** [31]).

In the classic UPPAAL, the model checker explores the model's state space based on the semantics of the model. In our extension of UPPAAL STRATEGO, an external library counsels the model checker to choose the right controllable actions based on the synthesized strategy. Specifically, the external library searches the strategy and returns the actions with the highest score at the current state. Consequently, the model checker only explores the state-action pairs that are returned from the external library. In this way, the agent model is under the control of the strategy without introducing the new model of *conductor*. In our extension of UPPAAL STRATEGO, Query (5.4) is changed to the following one, where $\sigma$ is a strategy, and a keyword *under* is used to instruct the model checker to call the external library containing a strategy named $\sigma$ when running the verification:

$$\texttt{A}\Diamond\ \phi\ \texttt{under}\ \sigma \tag{5.5}$$

Query (5.5) is initially proposed in the literature [30]. However, it can only be used to verify strategies that are synthesized by the graph-search based method in UPPAAL TIGA. In this dissertation, we extend the model checker in UPPAAL STRATEGO to verify a subset of strategies that are synthesized via reinforcement learning, against properties in the form of Query (5.5). MCRL can synthesize correctness-guaranteed mission plans for more than five agents working in environments with more tasks and milestones, which improves TAMAA and meets the requirement of our industrial use cases. In this dissertation, we prove that the new version of MCRL is *sound* (**Paper E** [31]). However, as the method is based on random simulation, it is not *complete* because it cannot guarantee to terminate with a mission plan being found even if such a plan exists. We leave this as a direction of future work.

### 5.2.3 Mission Planning in Stochastic Environments

When the environment becomes stochastic, the ending time of actions is decided by the environment stochastically. Therefore, the mission-planning problem now becomes a $1\frac{1}{2}$-player game. The goal of winning the game is to synthesize a strategy that owns the highest probability of finishing the agents' tasks while satisfying other requirements. As MCRL uses random simulation to sample traces for learning, the method is naturally suitable for solving the $1\frac{1}{2}$-player games.

In **Paper D** [32], we report our application of MCRL on a stochastic environment of an industrial use case. As aforementioned in Section 5.1, we reuse the HA models of agents that are modeled for the dynamic layer of our two-layer framework, and estimate the traveling time of agents, which may encounter moving objects stochastically. Next, we use the estimation of traveling time in the movement TG of agents, and run MCRL to obtain a strategy that is stochastically optimal. Besides, we can conduct analysis on the synthesized strategies, such as the *bottleneck analysis*, which shows at which milestone the agents stay idle for the longest time, and the *re-planning* when the probability distribution of the occurrence of pedestrians changes.

### 5.2.4 Correctness-Guaranteed Mission Plan Compression

Although MCRL can deal with larger numbers of agents, the mission plans in these cases usually take large memory space, which is time-consuming with respect to looking for the right action at each state of the model. In some applications, it is simply impossible to store plans that take too much memory space, such as in Airborne Collision Avoidance System X [57]. Hence, in this dissertation, we propose a method for compressing mission plans, while preserving the properties that are satisfied by the original plans. The method is called *MoCReL* (Model-checked Compressed Reinforcement Learning) [31], which is extended from MCRL.

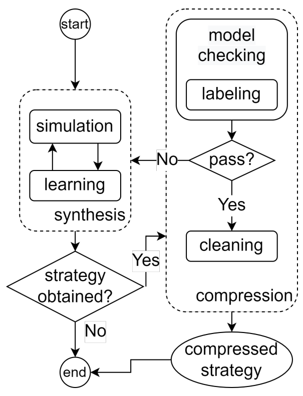


Figure 5.9: Workflow of MoCReL

Fig. 5.9 shows the workflow of this method. Similar to MCRL 2.0 in Fig. 5.8b, MoCReL starts with an iterative process of simulation and learning, which initially synthesizes a strategy. Next, instead of purely model checking a liveness property in the form of Query (5.5) against the strategy, MoCReL runs the model checking while labeling the visited state-action pairs in the strategy. Next, when the model checking passes, unlabeled pairs are cleaned from the strategy, which saves a massive portion of the original one. In **Paper E** [31], we prove that the method of synthesis and compression is sound in the sense that if a strategy is synthesized and compressed by MoCReL, it is guaranteed to be correct, that is, satisfying the liveness property that we aim to achieve in the mission planning problem. The experiments of MoCReL on our industrial use case show that the compression can save up to 99.95% memory space of what the original strategies take.

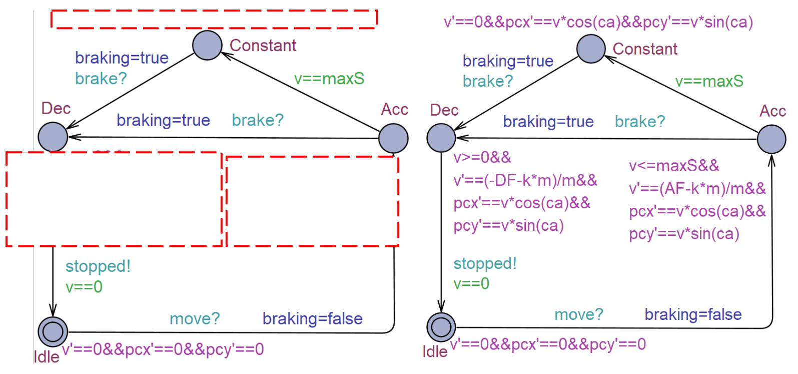
In summary, the contributions of TAMAA, MCRL, and MoCReL collectively address **Subgoal 3**.

## 5.3 Model-Checking Reach-Avoid Requirement of Nonlinear Agents

In this section, we describe the modeling of the agents' continuous motions and their embedded control systems, before the introduction of the two solutions for model checking the reach-avoid requirement of nonlinear agents.

### 5.3.1 Modeling Nonlinear Agents

Mission plans are considered to be static because they only take into account static milestones, obstacles, and tasks. When the agents start to execute mission plans, their controlling systems as well as the sensors and actuators must cooperate to function correctly. Moreover, the agents may encounter unexpected moving obstacles, and thus they must avoid the obstacles dynamically.



(a) The skeleton of the pattern (b) The HA of the pattern

Figure 5.10: The pattern of the linear motion component of an agent

In the two-layer framework, the agents' embedded controlling systems and their dynamic collision avoidance functions are modeled in the dynamic layer, where the modeling language is hybrid automata (HA). The HA models are introduced in **Paper A** [18], where we use patterns to design the templates of the models so that they can be reusable. Fig. 5.10a depicts the skeleton of the pattern for modeling the agents' movement, where locations represent the driving modes of the agent, continuous variables `v`, `pcx`, and `pcy` represent the agent's velocity, and its coordinate on the X and Y axes, respectively. Based on the Newtonian laws of motion, the template HA of this pattern is designed as shown in Fig. 5.10b, where the derivatives of `v`, `pcx`, and `pcy` are described by ordinary differential equations (ODE), which are presented as replaceable modules in the pattern (blank boxes in Fig. 5.10a).

The pattern-based design is also used in the HA templates of the agents' rotating motions and the models of their embedded control systems. As UPPAAL does not support hierarchical or recursive modeling, when the systems contain hierarchical structures such as processes and threads, functions and subfunctions, the models of the systems become very complex and hard to build and organize. The pattern-based method facilitates the building of the models by reusing the common components.

*Verification of the HA models.* After the HA models are constructed, the target now is to verify them against certain requirements. In this study, we are interested in a specific one called *reach-avoid requirement*: the agents must travel safely from their initial area to the goal area without a collision on the way. As depicted in Fig. 5.11, an agent starting from the initial area wants to follow its planned path and go to the goal area. Although the planned path avoids the static obstacles and reaches

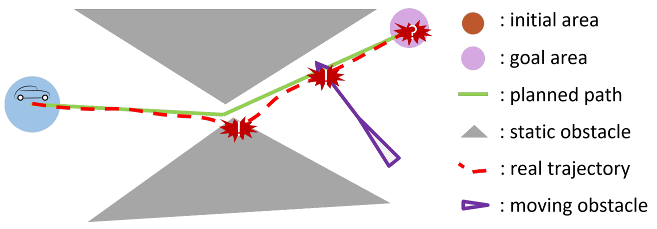


Figure 5.11: A scenario of agents violating the reach-avoid requirement even though the planned path is correct.

the goal area, the agent does not necessarily always satisfy the reach-avoid requirement. The challenge is two-fold: (i) the planned path is not practically tractable because of the sharp turnings at the waypoints where the linear line segments are connected, which generates inevitable tracking errors; (ii) the trajectory of an unpredictable moving obstacle can overlap with the agent's real trajectory, which possibly causing a collision. Moreover, even without considering challenges (i) and (ii), model checking HA is an undecidable problem [16]. We propose two approaches to solve this problem: (a) statistical model checking, and (b) exhaustive model checking after a provably correct transformation from the original HA model to an over-approximation as a discrete-time model, whose model checking is decidable.

### 5.3.2 Solution A: Statistical Model Checking

In **Paper A** [18], we propose the first solution, which uses statistical model checking (SMC) and UPPAAL SMC [33] as the tool. SMC randomly simulates the HA models instead of exhaustively exploring their state space. The random simulation samples a certain number of traces and checks if they satisfy the requirements. The ratio of the satisfactory traces among the samples is the probability of satisfaction of a requirement. Hence, instead of qualitative answers: true or false, SMC can return quantitative answers of probabilities. For instance, we check properties in the following forms[1], where constant numbers in the square brackets (e.g., [<= 70]) are used to define the simulation time, a Boolean variable `arrived` indicates whether the agent has arrived at the goal area or not, a clock variable `counter` counts the entire traveling time, and a Boolean variable `collided` indicates whether collision happens or not:

$$\texttt{Pr[<=70]}(\Diamond\ \texttt{arrived}\ \&\&\ \texttt{counter<=60}) \tag{5.6}$$

$$\texttt{Pr[<=110]}(\Box\ \texttt{!collided}) \tag{5.7}$$

Query (5.6) returns the probability of eventually reaching the goal area within 60 time units, and Query (5.7) returns the probability of always traveling with no collision. Moreover, we can use an UPPAAL SMC query of the form below, to sample variables such as the positions of the agent and moving obstacles, at different time points.

$$\begin{aligned}\texttt{simulate 1[<=110] }\{&\texttt{pcx,pcy,ocx[0],ocy[0],}\\ &\texttt{ocx[1],ocy[1],ocx[3],ocy[3]}\}\end{aligned} \tag{5.8}$$

The keyword *simulate* uses Monte-Carlo simulation to sample traces in the model state space. The constant number following *simulate* (e.g., 1) indicates the simulation rounds. Arrays `ocx` and `ocy` in Query (5.8) represent the positions of moving obstacles at x and y axes. In this dissertation, we choose a novel collision avoidance algorithm based on dipole flow fields [38] for the dynamic collision avoidance function of the agents, and simulate the HA models by using Query (5.8). The trajectories obtained from the query are shown in Fig. 5.12, where "A" and "B" are two predefined moving obstacles, and "C" is a dynamically generated obstacle that moves "recklessly" towards the agent, so the latter turns around to avoid the obstacle.

[1]Properties in Section 5.3.2 are supported by UPPAAL SMC 4.1.24.

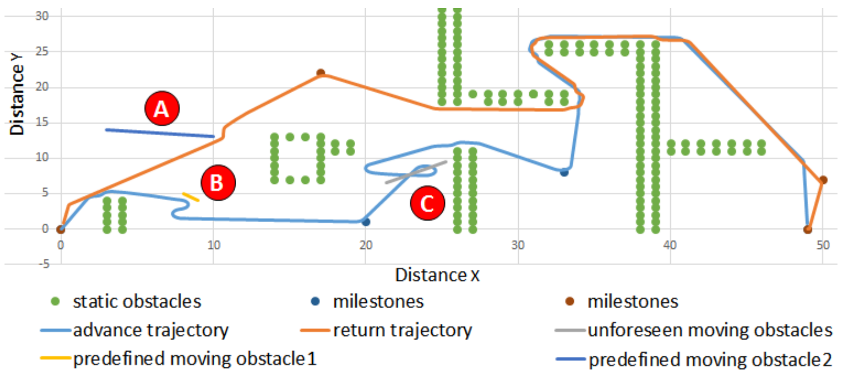


Figure 5.12: The trajectory of an agent in a map with moving obstacles

In summary, by using statistical model checking in UPPAAL SMC, we can get the probabilistic answers of satisfying the reach-avoid property and simulate the agent models to get the samples of movement. When the occurrence of moving obstacles are stochastic, this approach addresses **Subgoal 4**. When the moving obstacles are non-deterministic in the sense that they can non-deterministically appear at any moment during the verification time, from any position in the map, travel at any speed and angle within their capabilities, solution A is not enough. Hence, we propose *solution B* for this case.

### 5.3.3 Solution B: Exhaustive Model Checking

If we can enumerate all the possible behaviors of the non-deterministic moving obstacles and exhaustively check if the agent model satisfies the reach-avoid property when such obstacles do exist, then we can get an absolute answer to the reach-avoid problem. However, HA models have infinite states, which makes model checking such models an undecidable problem. Although TA models are infinite-state models too, because of the continuous clock variables, with a certain method of abstraction, e.g., zones [52], the states of TA models are represented symbolically and thus the state space becomes finite, so the problem of model checking TA becomes decidable. Unfortunately, there is no such an abstraction technique for nonlinear HA models [17].

Therefore, in this dissertation (**Paper F** [37]), we propose a novel transformation from the nonlinear trajectories of agents to discrete-time trajectories and prove that, under certain assumptions, if the latter satisfies the reach-avoid property, the former must also satisfy the property. The assumptions include: (i) the tracking errors of agent have a Lyapunov function [12], and (ii) the sampling periods (denoted by $\epsilon$) of the discrete-time trajectory satisfy: $\epsilon \leq \frac{L}{\|\mathbf{V}\|}$, where $L = L_a + L_o$, where $L_a$ is the tracking-error boundary of the agent, $L_o$ is the smallest tracking-error boundary among dynamic obstacles[2], and $\mathbf{V}$ is the maximum linear velocity of the moving obstacles.

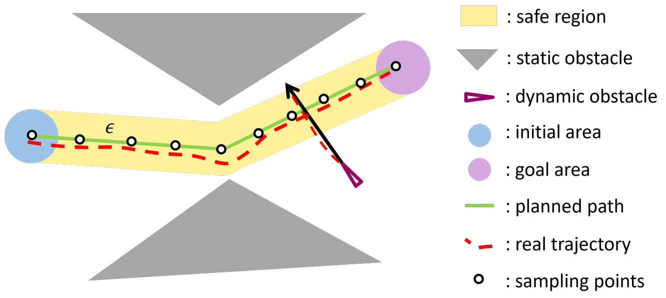


Figure 5.13: Safe region of a planned path and its discrete-time path

The conclusion reached by Fan *et al.* [58] shows that if the agents' tracking errors have a Lyapunov function, their real trajectories are bounded within a safe region centered by the planned path (as shown in Fig. 5.13). Based on this result, we further introduce dynamic obstacles and prove two theorems (Theorem 1 and Theorem 2) in **Paper F** [37], which show that by sampling on the planned path with a certain length of periods, we can obtain a discrete-time trajectory that preserves the reach-avoid property of the real nonlinear trajectory. Thus, the proven transformation results in a discrete-time over-approximating model of trajectories, whose model checking is decidable. Based on the model transformation, we propose a tool-supported technique of model checking nonlinear agents. We use UPPAAL STRATEGO [30] as the model checker, since its latest version supports calling external functions, which enables us to implement collision avoidance algorithms as external libraries.

[2]When no dynamic obstacle is detected, $L_o$ is zero.

Fig. 5.14 shows the workflow of the verification approach. The input of the approach are the parameters of the agents (i.e., autonomous vehicles and dynamic obstacles) and their boundary of the tracking errors, as well as the environment (e.g., static obstacles). In Step 1, users provide their nonlinear vehicle models, which are used for calculating the boundary of tracking errors. This module is the approach provided by Fan *et al.* [12], which is not the focus of this dissertation. We use the output of their approach in our models for verification. In Step 2, users configure the parameters of the approach, which are used for instantiating the UTA templates of the discrete-time models in Step 3. Note that the user-programmed collision-avoidance algorithm is embedded in the models as executable libraries, e.g., Dynamic-Link Libraries (DLL) in Windows, or Shared Objects (SO) in Linux.

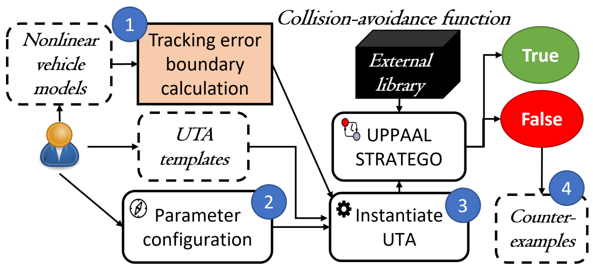


Figure 5.14: The workflow of solution B for verifying the reach-avoid property against nonlinear vehicle models

After the instantiation of UTA, the model checker verifies the model by traversing its state space, calling the external library at the transitions where external functions are invoked, and checking if the vehicle model avoids all obstacles and reaches the destination under all circumstances. Since the model checker only uses the output of the external functions in the verification, the external library is treated as a black box whose implementation detail is not concerned by the verification. If the verification result is "true", the agents are guaranteed to satisfy the reach-avoid requirement under the current parameter configuration; otherwise, counter-examples are returned by the model checker for the users to debug their algorithm or change the configuration of the parameters (Step 4).

To demonstrate the performance of our approach, in this dissertation, we program the collision-avoidance algorithm based on dipole flow fields [38] as an external library and verify the model that is linked to this library. The results identify two problematic scenarios where collision happens inevitably. Using these counter-examples, we improve the algorithm and verify the model again. The new verification results show that the algorithm enables the agents to travel safely and reach their destination in an environment with one moving obstacle, and the verification time is in the range of several seconds or minutes. Details of the experiments and the results are in **Paper F** [37]. In summary, the experimental results demonstrate the ability of the approach for finding bugs in the design stage of a collision-avoidance algorithm and demonstrate the absence of certain bugs in an improved version of the algorithm. The contributions of solutions A and B collectively address **Subgoal 4**.

## 5.4 Validation on Industrial Systems

As the dissertation aims to deal with industrial systems, our methods need to be automated by tools so that engineers can benefit from formal methods without the required expertise. In this dissertation (**Paper C** [21]), we propose *MALTA*, a tool-supported methodology of mission planning. Fig. 5.15 depicts the structure of MALTA, which is built of

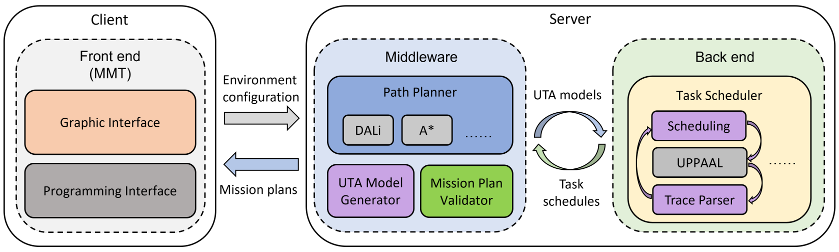
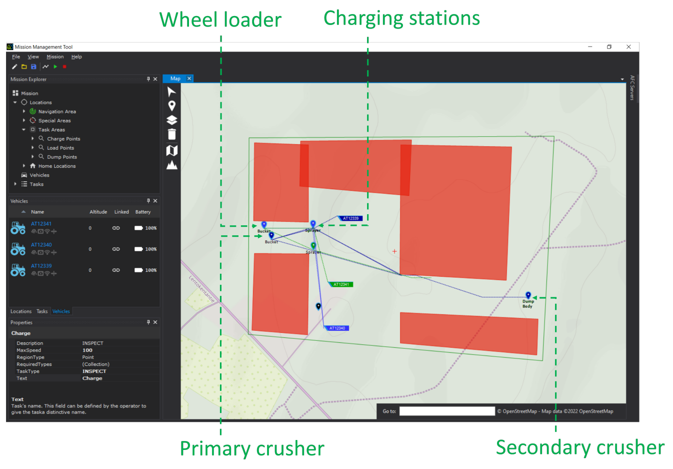
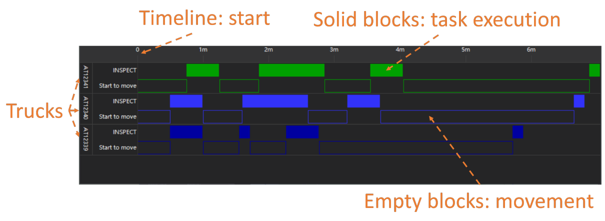


Figure 5.15: The structure of MALTA

three parts: a *front end* providing user-interface, a *middleware* providing path planning and building mission plans from paths and schedules, and a *back end* dedicated for task scheduling. The toolset adopts a Client-Server architecture. The reason is twofold: first, the front end of the toolset is a GUI that had been independently designed before the mid-



(a) Path finding of the example in MALTA



(b) Task scheduling of the example in MALTA

Figure 5.16: An example of mission planning in MALTA

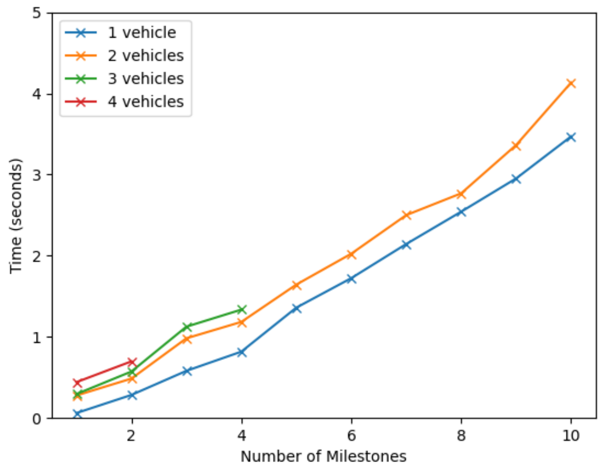
dleware and the back end were designed. Beside the GUI, the front end also provides a group of programming interfaces and data structures for communication. Therefore, the front end is open for extension without requiring code updates. Second, the computation of mission plans can be quite expensive. As we show in Section 5.2, synthesizing mission plans for multiple agents can cost hours on a computationally powerful server. Therefore, a separation of the front-end GUI from the function of mission plan synthesis allows the users to move computation to a dedicated server, which is a user-friendly and efficient pattern of design.

In the front end (see Fig. 5.16a), users can configure their environment including the navigation areas, milestones, tasks, agents, etc., after which, the environmental configuration is transferred to the middleware. The middleware receives the environment configuration and passes it to a path planner. Any path-finding algorithm that supports the desired environmental constraints can be used.

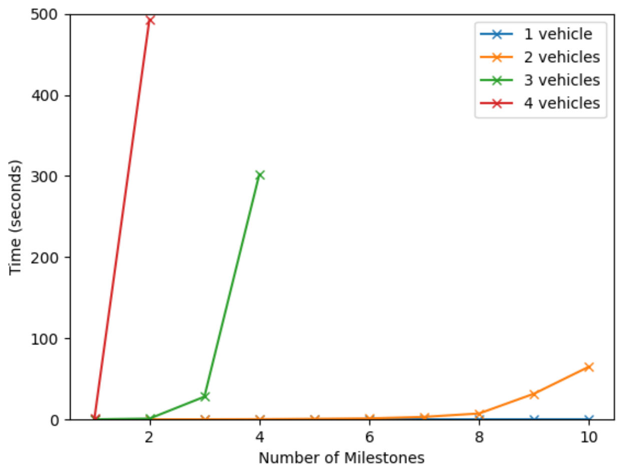
In this dissertation, we include several path-finding algorithms: A* [22], Theta* [23], and DALi [24] and two of its improved versions. A* algorithm is a classic algorithm that discretizes the environment as a Cartesian grid and computes the shortest paths between two points in the grid. The resulting paths are restricted to the edges and diagonals of the cells of the grid. Theta* algorithm is based on A*, which is not restricted to only use edges and diagonals of cells, and thus its resulting paths are smooth in the sense of having as few sharp turnings as possible. The DALi algorithm and its two improved versions are special path-finding algorithms in complex environments. The algorithm takes into account environmental constraints and user's preferences, such as temporary forbidden areas and hot maps. The resulting paths of DALi avoid the temporary forbidden areas when necessary and stick to the user-preferred areas as close as possible.

After paths are found, the middleware generates UTA models as we have described in Section 5.2. These UTA models are transferred to the back end, where TAMAA is executed to synthesize task schedules. The path-finding module in the middleware and task-scheduling module in the back end run iteratively until all the environmental constraints are met by the paths and task schedules. Finally, mission plans are generated based on the synthesized paths and task schedules and then shown in the front end (see Fig. 5.16a).

We evaluate the toolset in an industrial use case: an autonomous quarry that contains autonomous wheel loaders and trucks that need to dig and transport a certain amount of stones every day. The quarry use case is provided by Volvo Construction Equipment in Sweden. The evaluation results show the computation time of various versions of the path-finding algorithm and task scheduler in different sizes of environments. As shown in Fig. 5.17a, the computation time of path finding increases linearly with the increased numbers of agents and milestones. It is because the path-finding algorithm is executed independently in each of the agents and the resulting paths do not need to consider each other. The overlapping of the paths is dealt with by the dynamic layer of our two-layer framework. However, as task scheduling uses the composed model of all agents, the computation time increases exponentially



(a) Path finding time of DALI wrt the numbers of agents and milestones



(b) Task scheduling time wrt the numbers of agents and milestones

Figure 5.17: Computation time of MALTA

Table 5.3: Contribution of included papers with respect to research goals

| | Subgoal 1 | Subgoal 2 | Subgoal 3 | Subgoal 4 | Subgoal 5 |
|---|---|---|---|---|---|
| Paper A | X | | | X | |
| Paper B | | X | X | | X |
| Paper C | | | X | | X |
| Paper D | X | | X | | |
| Paper E | | | X | | |
| Paper F | | | | X | X |

with the linearly increased number of agents (as shown in Fig. 5.17b), which complies with the experimental results in **Paper B** [26]. More experiments and the results are reported in **Paper C** [21].

As aforementioned, to meet the requirement of agent numbers in industrial systems, we propose a scalable method of mission planning called MCRL. In the experiments of **Paper B** [26], the method is adopted in the toolset MALTA. In summary, we develop a tool that automates the mission planning methods for different kinds of environments. The tool facilitates the model building and mission plan synthesis by using formal methods. This contribution addresses **Subgoal 5**.

## 5.5 Research Goals Revisited

In this section, we present the technical contributions of this dissertation 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 5.3.

- **Paper A** [18] proposes a two-layer framework for the formal modeling and verification of autonomous agents, such that designers can utilize formal methods to analyze and design autonomous agents via a systematic approach that is founded on rigorous design elements based on formal models. **Paper D** [32] shows the communication between the static layer and dynamic layer of the framework. These contributions address **Subgoal 1**: *Provide a means that decouples the design of mission planning from the agents' autonomous operation in a continuous dynamic environment, supported by model checking techniques.*

- **Paper B** [26] introduces our mission-planning algorithms in different types of environments, and analyzes their weaknesses and strengths. This contribution addresses **Subgoal 2**: *Assuming different types of environments of mission planning, that is, deterministic and collaborative ones, stochastic ones, or non-deterministic ones, identify suitable methods for mission planning, and evaluate their appropriateness in various scenarios.*

- **Paper C** [21] introduces the approaches for mission planning in a deterministic environment with complex road conditions. **Paper B** [26] shows the second version of MCRL integrated into UPPAAL STRATEGO, and demonstrates the performance of the algorithm when dealing with relatively large numbers of agents. **Paper D** [32] is about the probabilistic mission planning, and **Paper E** [31] extends MCRL with strategy compression. These contributions address **Subgoal 3**: *Provide scalable approaches that involve model checking, for mission-plan synthesis and compression of MAS, such that the computation time is reasonable when the number of agents is large and the resulting mission plans only contain the useful information that guarantees the agents to satisfy given temporal requirements.*

- **Paper A** [18] uses statistical model checking to verify the HA models of agents against the reach-avoid requirement when facing unforeseen moving obstacles. **Paper F** [37] proposes our the model transformation from nonlinear trajectories to discrete-time trajectories and uses symbolic model checking to give a qualitative answer of true or false to the reach-avoid verification. These contributions address **Subgoal 4**: *Ensure the reach-avoid requirement of agents when executing mission plans in environments with unforeseen static and dynamic obstacles.*

- By implementing our mission-planning approaches in a toolset named *MALTA*, a tool-supported approach for reach-avoid verification, and applying our proposed methods to the industrial use case of an autonomous quarry, **Paper B** [26], **Paper C** [21], and **Paper F** [37] collectively address **Subgoal 5**: *Develop automated support that integrates the approaches of mission plan synthesis and verification, and assesses the applicability of the approaches in an industrial use case.*

These contributions together provide methodologies for mission planning and reach-avoid verification of MAS, supported by formal methods and tools, which address our overall research goal.

# Chapter 6

# Related Work

In this chapter, we present some of the related work in mission planning and verification of agents. These previous studies pave the way of mission plan synthesis and reach-avoid verification by using formal methods, which inspire our work. 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 has motivated us to extend the study and fill the research gaps.

## 6.1 Multi-Layer Frameworks for Agent Design and Verification

Belta *et al.* [59] 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 [60, 61]. Bhatia *et al.* [62, 63] 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.* [64, 65] propose their method for motion planning of multiple-agent systems using various temporal logics. Saddem *et al.* [66] 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. However, few of the studies have decoupled the problems of mission planning and reach-avoid verification in different layers, where suitable formal methods are adopted, respectively. In addition, our communication between the layers is bidirectional. Not only mission plans can be sent to and verified in the dynamic layer, but once new road conditions are detected, e.g., new temporary obstacles, the environmental information is also sent back to the static layer for re-planning. All these aspects of our framework are supported by formal methods combined with state-of-the-art approaches of synthesis, e.g., Q-learning [51], and collision avoidance, e.g., the algorithm based on dipole flow fields [38].

## 6.2 Mission Planning of Agents

Path finding in the Artificial Intelligent (AI) community, has been a research interest since the early days of robotics [67]. Sampling-based methods like Rapidly-exploring Random Tree (RRT) [68] and a method based on probabilistic roadmaps [69], and graph-search-based methods like A* [22] and Theta* [23], are two typical branches of path-finding algorithms. These algorithms are dedicated to find collision-free paths in a confined environment, interspersed with static obstacles. However, when the environment starts to be dynamic, e.g., moving obstacles are involved, or the agents need to interact with it, e.g., picking an object and carrying it to another position, the path-finding algorithms become insufficient. Alami *et al.* [70] and Hauser *et al.* [71] propose a modal structure of the robots and their working environments. Since the switch of modes is discrete, the problem is about identifying the modes of the systems, defining the transitions among the modes, and traversing the state space in order to find a trace that satisfies some certain constraints. This is the so-called task planning in the AI community [72].

In recent decades, there has been a growing interest in task planning with complex goals. There is an important line of work of task planning that uses temporal logic to specify the high-level requirements of tasks [73, 74]. Linear Temporal Logic (LTL) is one of the most widely used logic for requirement specification [75, 76, 62], because of its expressive power that is able to capture relatively complex requirements. Different from these studies, we adopt Timed Computation Tree Logic (TCTL). (T)CTL and LTL are subsets of a temporal logic family named CTL* [6]. Each of the logics has an expressive power that is orthogonal to the other, hence they are used in different problems, respectively. TCTL enables one to express timing requirements, e.g., digging 1000 $m^3$ of stones within 24 hours, which is of high industrial concern, in an attempt to ensure productivity when using autonomous vehicles. Most importantly, our MCRL method innovatively combines model checking with reinforcement learning to deal with a large number of agents in a composed model while preserves the correctness guarantee of the results.

In the field of combining formal methods with reinforcement learning (RL), Behjati *et al.* [77] attempt to solve the state-space-explosion problem of model checking LTL properties by using RL. The method proposed by Bouton *et al.* [78] enforces probabilistic guarantees on agents during the course of reinforcement learning. Jothimurugan *et al.* [79] propose *DIRL*, a synthesis approach that interleaves Djikstra's algorithm with RL to train agents. Brázdil *et al.* [80] provide learning algorithms for searching MDP (Markov Decision Processes) to verify various reachability properties. Legay *et al.* [81] present a scalable approach of verification for MDP. In comparison, Our MCRL combines reinforcement learning with model checking in another direction, that is, using reinforcement learning to alleviate the state-space explosion problem of model checking.

Some studies have proposed tool-supported methods of planning by using UPPAAL and its branches. Andersen *et al.* [82] present a UPPAAL-based method for motion planning of multi-robot systems. Their method uses reachability queries to generate motion plans, which is not sufficient for synthesizing comprehensive strategies that consider time intervals as the execution time of motions. Bersani *et al.* [11] present the PuRSUE (Planner for RobotS in Uncontrollable Environments) approach, which supports users to configure their robotic applications and automatically generate their controllers by using UPPAAL TIGA. However, their method can only deal with 2 robots simultaneously. Comparing with these studies, we formally formulate the planning problem for multi-agent systems and categorize the problem according to the types of the environment. For different type of environment, we propose different solutions and analyze their strengths and weaknesses. In a nutshell, our solutions for the mission-planning problem is systematic and scalable in realistic industrial scenarios.

In the area of strategy compression, Julian *et al.* explore several ways of compressing strategies by using origami compression [83] or deep neural network [57][84]. Ashok *et al.* propose a decision-tree-based method for concisely representing strategies [85][86]. Their tool named *dtControl* is able to compress strategies produced by UPPAAL TIGA. Compared with these methods, the strategy compression in MoCReL focuses on cleaning the unused data in the strategies rather than representing them in different forms. Compression in MoCReL that relies on exhaustive model checking inherently provides correctness guarantee of the results, which needs extra effort to achieve in other methods [84]. Piterman *et al.* minimize strategies by removing redundant states [87]. Intuitively, states are considered redundant if the corresponding strategies satisfy the desired properties with and without these states. In contrast, the unused state-action pairs that MoCReL removes are those that are explored by reinforcement learning but do not satisfy the properties.

## 6.3 Verification of Agents

The reach-avoid verification is always a focus of research in the Robotic community. Automata-based methods have been studied widely in solving this problem [88, 89, 8, 60]. In these papers, the authors study agents that autonomously carry out tasks like searching for an object, avoiding an obstacle, and missions sequencing. The main method of verification that is used in these studies is model checking. As aforementioned, when the environment contains unforeseen obstacles, or the dynamics of the agents becomes continuous or even nonlinear, classic model checking is not able to solve the problem.

Runtime verification that monitors the behavior of autonomous systems addresses the above-mentioned shortage to some extent [90][91][92][93]. 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.

Other formal methods such as theorem proving have also been investigated in conducting the reach-avoid verification of agents. Mitsch *et al.* [94] propose a method to verify safety properties of robots. Their method is based on hybrid system models and differential dynamic logic for theorem proving in KeYMaera. Abhishek *et al.* [95, 96] also use

KeYMaera for collision-avoidance verification. Their models consider the realistic geometrical shapes of vehicles, as well as the combination of maneuvre and braking. Heß *et al.* [97] propose a method to verify an autonomous robotic system during its operation, in order to cope with changing environments. O'Kelly *et al.* [98] have developed a verification tool, called APEX, and investigated the combined action of a behavioral planner and state lattice-based motion planner to guarantee a safe vehicle trajectory. Our work differs from the above studies in the following aspects: based on the bounded tracking errors of actual trajectories, we prove that the reach-avoid verification of nonlinear vehicle models can be simplified to a decidable problem of verifying over-approximations in form of discrete-time models. Additionally, we consider the difficulties of using formal methods in industrial scenarios and propose methods such as pattern-based modeling and external libraries embedded in formal models to facilitate engineers to benefit from formal methods with a lesser effort of learning.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

In this dissertation, we research the mission planning and reach-avoid verification of autonomous agents by employing formal methods. To achieve separation of concerns, we decouple the discrete mission planning from the reach-avoid verification of the mission plan execution in a continuous environment, by proposing a two-layer framework that consists of a static layer and a dynamic layer. The static layer contains the milestones, tasks, and static obstacles, and is dedicated to the mission planning problem. The continuous movement of agents, as well as the unforeseen moving obstacles are considered in the dynamic layer. In the static layer, we propose different methods of mission planning for different types of environments. For the environment that is fully controlled by agents, we design a correctness-guaranteed synthesis method based on model checking, namely TAMAA. The method is capable of synthesizing the fastest mission plans and guarantees its results to satisfy various requirements. If the environment non-deterministically decides the ending time of agent actions, we adapt the TA models of agents to be TG models. We first adopt TAMAA in UPPAAL TIGA for synthesizing mission plans based on the TG models. Our experiments show that both TAMAA in UPPAAL and UPPAAL TIGA have a limitation on the number of agents that they can handle. To scale the solution to a larger number of agents, we propose a novel approach called MCRL, which combines model checking with reinforcement learning. The new method is able to cope with more agents than TAMAA. In addition, MCRL is demonstrated to be able to provide stochastically optimal solutions of mission planning when the environment contains probabilistic information about obstacles. Moreover, we design a method of mission plan compression based on MCRL, namely MoCReL. The mission plans that are synthesized and compressed by MoCReL are guaranteed to be correct and use down to 0.05% of the memory space of the original ones.

For the dynamic layer of the framework, we propose a pattern-based method of designing the models described by hybrid automata (HA), which can generate the continuous movement of agents and moving obstacles. As model checking HA models is undecidable, we propose two ways of the reach-avoid verification: statistical model checking, and exhaustive model checking with a 2-step model transformation that we prove correct, which results in a discrete-time over-approximated model that can be model checked. First, when the occurrence of moving obstacles is stochastic, we employ statistical model checking to provide probabilistic results of verification. Second, when the moving obstacles appear and behave non-deterministically, we transform the continuous trajectories to discrete-time ones and prove that if the latter satisfy the reach-avoid property, the former must also satisfy it. Based on the model transformation, we propose a tool-supported model checking technique in UPPAAL STRATEGO. The experimental results show that the approach is able to detect problematic scenarios for an improved version of the collision-avoidance algorithm based on dipole flow fields, where collisions with moving obstacles are inevitable, and demonstrate the absence of bugs in the agents when such scenario does not exist. These two verification approaches provide solutions for different types of environments.

To facilitate the use of our methods by engineers in industrial applications, we develop a toolset called MALTA and a pattern-based method of modeling and verification. MALTA provides a GUI for configuring the environment and missions of the agents, and is open for extensions of path-finding and task-scheduling algorithms. With the help of MALTA, engineers can benefit from using formal methods for mission planning without the corresponding expertise. The pattern-based modeling facilitates the model construction. When facing different agents with different kinematics, users of our method only need to replace certain modules of the patterns to create their own agent models. Especially when the embedded control systems of the agents have hierarchical structures, the patterns alleviate the complexity of modeling in UPPAAL, which does not support multi-level models.

## 7.2 Future Work

The future work has several possible directions. One is to enhance the communication between the two layers of the framework so that they communicate in a real-time manner, such that the mission planning and verification can be optimized. Another direction is about investigating the combination of reinforcement learning and model checking in the synthesis problem. So far, reinforcement learning aids model checking because the former enables the latter to verify mission plans of large numbers of agents. However, when the goal of the agents is a rare event in the sense that random simulation can hardly reach the goal state, reinforcement learning finds it very hard to obtain results. A counter-example-guided approach can complement reinforcement learning by providing counter-examples of incomplete mission plans and accelerate the learning phase. In the direction of scalable synthesis for MAS, we want to investigate the methods of decoupling the system into smaller and more tractable sub-systems, while still achieving their common goal of the mission.

In the direction of agent verification, applications with more complex kinematics and dynamics of agents may be investigated. How to adapt techniques that can be beneficial for engineers, such as barrier certificate or bounded model checking, falls also within our research interest.

# 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. Springer.

[2] Đorđe Petrović, Radomir Mijailović, and Dalibor Pešić. Traffic accidents with autonomous vehicles: type of collisions, manoeuvres and errors of conventional vehicles' drivers. *Transportation research procedia*, 45:161–168, 2020. Elsevier.

[3] Walter F Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998. IEEE.

[4] Philip Koopman and Michael Wagner. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intelligent Transportation Systems Magazine*, 9(1):90–96, 2017.

[5] 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.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

[7] Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D Ames, Jessy W Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. Correct-by-construction adaptive cruise control: Two approaches. *IEEE Transactions on Control Systems Technology*, 24(4):1294–1307, 2015.

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

[9] 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.

[10] 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.

[11] Marcello M Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione, and Matteo Rossi. Pursue-from specification of robotic environments to synthesis of controllers. *Formal Aspects of Computing*, 32(2):187–227, 2020. Springer.

[12] Chuchu Fan, Kristina Miller, and Sayan Mitra. Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In *International Conference on Computer Aided Verification*, pages 629–652. Springer, 2020.

[13] Rong Gu, Eduard Enoiu, and Cristina Seceleanu. Tamaa: Uppaal-based mission planning for autonomous agents. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 1624–1633, 2020.

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

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

[16] Kārlis Čerāns. *Algorithmic problems in analysis of real time system specifications*. PhD thesis, University of Latvia, 1992.

[17] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of computer and system sciences*, 57(1):94–124, 1998. Elsevier.

[18] 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.

[19] 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.

[20] 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. Springer.

[21] Rong Gu, Eduard Baranov, Afshin Ameri, Eduard Enoiu, Baran Cürüklü, Cristina Seceleanu, Axel Legay, and Kristina Lundqvist. Synthesis and verification of mission plans for multiple agents under complex road conditions. *Submitted to Transactions on Software Engineering and Methodology (TOSEM)*, 2022. ACM.

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

[23] 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. AI Access Foundation.

[24] Alessio Colombo, Daniele Fontanelli, Axel Legay, Luigi Palopoli, and Sean Sedwards. Efficient customisable dynamic motion planning for assistive robots in complex human environments. *Journal of ambient intelligence and smart environments*, 7(5):617–634, 2015. IOS Press.

[25] Gerd Behrmann, Alexandre David, Emmanuel Fleury, Kim Larsen, Didier Lime, and Ecole Nantes. Uppaal-Tiga: Time for playing games! (tool paper). In *Proceedings of the 2007 Computer Aided Verification*. Springer, 2007.

[26] Rong Gu, Peter Jensen, Danny Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Verifiable strategy synthesis for multiple autonomous agents: A scalable approach. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–20, 2022. Springer.

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

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

[29] Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Verifiable and scalable mission-plan synthesis for autonomous agents. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 73–92. Springer, 2020.

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

[31] Rong Gu, Peter Jensen, Danny Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Strategy synthesis and compression for multi-agent systems. *Submitted to Journal of Science of Computer Programming (SCP)*, 2022. Elsevier.

[32] Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Probabilistic mission planning and analysis for multi-agent systems. In *International Symposium on Leveraging Applications of Formal Methods*, pages 350–367. Springer, 2020.

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

[34] Chuchu Fan, Umang Mathur, Sayan Mitra, and Mahesh Viswanathan. Controller synthesis made real: Reach-avoid specifications and linear dynamics. In *International Conference on Computer Aided Verification*, pages 347–366. Springer, 2018.

[35] Kai-Chieh Hsu, Vicenç Rubies-Royo, Claire J Tomlin, and Jaime F Fisac. Safety and liveness guarantees through reach-avoid reinforcement learning. *arXiv preprint arXiv:2112.12288*, 2021.

[36] Jaime F Fisac, Mo Chen, Claire J Tomlin, and S Shankar Sastry. Reach-avoid problems with time-varying dynamics, targets and

constraints. In *Proceedings of the 18th international conference on hybrid systems: computation and control*, pages 11–20, 2015. ACM.

[37] Rong Gu, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Model checking collision avoidance of nonlinear autonomous vehicles. In *International Symposium on Formal Methods*, pages 676–694. Springer, 2021.

[38] 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. Frontiers Media SA.

[39] Stephen Edwards, Luciano Lavagno, Edward A Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[40] Stephen C Kleene et al. Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41, 1956. Princeton, NJ.

[41] Carl Adam Petri. Communication with automata. *University of Hamburg*, 1966.

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

[43] 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.

[44] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *International Conference on Concurrency Theory*, pages 66–80. Springer, 2005.

[45] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In *International Symposium on Automated Technology for Verification and Analysis*, pages 81–97. Springer, 2019.

[46] Paul E Black. Manhattan distance. *Dictionary of Algorithms and Data Structures*, 18:2012, 2006. US National Institute of Standards and Technology. Retrieved May.

[47] 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. Springer.

[48] 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.

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

[50] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*. MIT Press, 2000.

[51] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. *King's College, Cambridge*, 1989.

[52] Gerd Behrmann, Patricia Bouyer, Kim G Larsen, and Radek Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *International Journal on Software Tools for Technology Transfer*, 8(3):204–215, 2006. Springer.

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

[54] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. *Citeseer*, 1995.

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

[56] 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.

[57] Kyle D Julian, Mykel J Kochenderfer, and Michael P Owen. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019. American Institute of Aeronautics and Astronautics.

[58] Chuchu Fan, Zengyi Qin, Umang Mathur, Qiang Ning, Sayan Mitra, and Mahesh Viswanathan. Controller synthesis for linear system with reach-avoid specifications. *IEEE Transactions on Automatic Control*, 2021.

[59] 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.

[60] 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. SAGE Publications.

[61] 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. SAGE Publications.

[62] 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.

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

[64] 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.

[65] 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. Elsevier.

[66] 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. Elsevier.

[67] Nils J Nilsson et al. Shakey the robot. *Articial Intelligence Center, SRI International Menlo Park, California*, 1984.

[68] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report*, 1998.

[69] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 1996.

[70] Rachid Alami, Thierry Simeon, and Jean-Paul Laumond. A geometrical approach to planning manipulation tasks. the case of discrete placements and grasps. In *The fifth international symposium on Robotics research*. MIT Press, 1990.

[71] Kris Hauser and Jean-Claude Latombe. Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, 2010. SAGE Publications.

[72] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[73] 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.

[74] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal methods for discrete-time dynamical systems*. Springer, 2017.

[75] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010.

[76] Mingyu Cai, Hao Peng, Zhijun Li, and Zhen Kan. Learning-based probabilistic ltl motion planning with environment and motion uncertainties. *IEEE Transactions on Automatic Control*, 2020.

[77] 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.

[78] 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.

[79] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional reinforcement learning from logical specifications. *Advances in Neural Information Processing Systems*, 34, 2021. MIT Press.

[80] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtěch Forejt, Jan Křetínskỳ, Marta Kwiatkowska, David Parker, and Mateusz Ujma. Verification of markov decision processes using learning algorithms. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2014.

[81] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Scalable verification of markov decision processes. In *International Conference on Software Engineering and Formal Methods*. Springer, 2014.

[82] 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. Elsevier.

[83] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2016.

[84] Kyle D Julian and Mykel J Kochenderfer. Guaranteeing safety for neural network-based aircraft collision avoidance systems. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2019.

[85] Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínskỳ, Maximilian Weininger, and Majid Zamani. dtcontrol: Decision tree learning algorithms for controller representation. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–7. ACM, 2020.

[86] Pranav Ashok, Mathias Jackermeier, Jan Křetínský, Christoph Weinhuber, Maximilian Weininger, and Mayank Yadav. dtcontrol 2.0: Explainable strategy representation via decision tree learning steered by experts. *arXiv preprint arXiv:2101.07202*, 2021.

[87] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 364–380. Springer, 2006.

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

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

[90] 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. Springer.

[91] 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. IEEE, 1990.

[92] 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.

[93] 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.

[94] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher, and André Platzer. Formal verification of obstacle avoidance and navigation

of ground robots. *The International Journal of Robotics Research*, 36(12):1312–1340, 2017. SAGE Publications.

[95] Aakash Abhishek, Harry Sood, and Jean-Baptiste Jeannin. Formal verification of braking while swerving in automobiles. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–11. ACM, 2020.

[96] Aakash Abhishek, Harry Sood, and Jean-Baptiste Jeannin. Formal verification of swerving maneuvers for car collision avoidance. In *2020 American Control Conference (ACC)*, pages 4729–4736. IEEE, 2020.

[97] Daniel Heß, Matthias Althoff, and Thomas Sattel. Formal verification of maneuver automata for parameterized motion primitives. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1474–1481. IEEE, 2014.

[98] Matthew O'Kelly, Houssam Abbas, Sicun Gao, Shin'ichi Shiraishi, Shinpei Kato, and Rahul Mangharam. Apex: Autonomous vehicle plan verification and execution. *SAE World Congress*, 2016. SAE.

# II

# Included Papers

# Chapter 8

# Paper A: Towards a Two-layer Framework for Verifying Autonomous Vehicles

Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist
Published in 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.

## 8.1 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 8.2, 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 8.3 describes the function of the autonomous wheel loader and its architecture. In Section 8.4, we present the conceptual two-layer framework, and in Section 8.5 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 8.6. We compare to related work in Section 8.7, before concluding and outlining future lines of research in Section 8.8.

## 8.2 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.

### 8.2.1 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:

$$HA =< L, l_0, X, \Sigma, E, F, I >, \tag{8.1}$$

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 ($\Sigma_i$) and outputs ($\Sigma_o$), $E$ is a finite set of edges of the form $(l, g, a, \varphi, l')$, where $l$ and $l'$ are locations, $g$ is a predicate on $\mathbb{R}^X$, $a \in \Sigma$ is an action label, and $\varphi$ 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 $u \models I(l)$, and transitions defined as: (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) discrete transitions ($< l, u > \xrightarrow{a} < l', u' >$ if edge $l \xrightarrow{g,a,r} l'$ exists such that $a \in \Sigma, u \models g$, 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 $u' \models I(l')$).

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]`($\psi_2$)) are also supported.

### 8.2.2 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].

### 8.2.3 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].

## 8.3 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 8.1, 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 Fig-

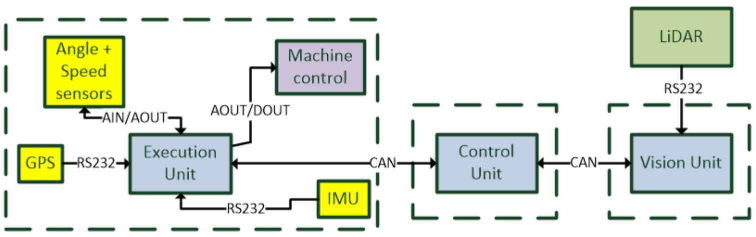


Figure 8.1: The architecture of the AWL's embedded control system

ure 8.2. 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 `Execution Function`, in which three sub-functions are called. The function `AdjustAngle` adjusts the

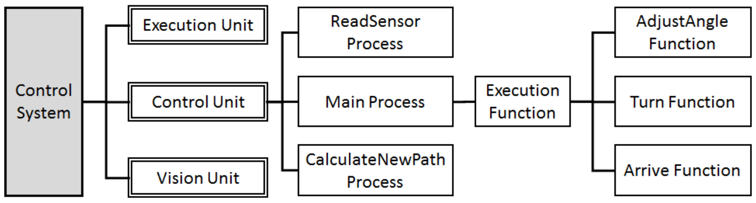


Figure 8.2: Process allocation in the control system

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 8.1, 8.2), 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.

## 8.4 A Two-level Framework for Planning and Verifying Autonomous Vehicles

As it is shown in Figure 8.3, 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 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

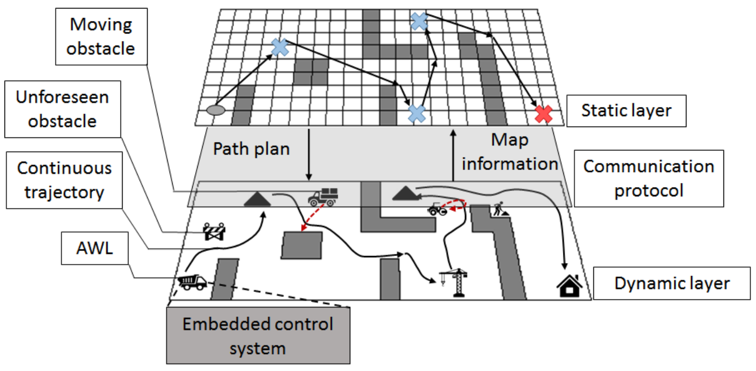


Figure 8.3: Two-layer framework for planning and verifying autonomous vehicles

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.

## 8.5 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 8.1, 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 unit, and motion sensors (GPS, IMU, Angle and Speed sensors) connecting to the execution unit.

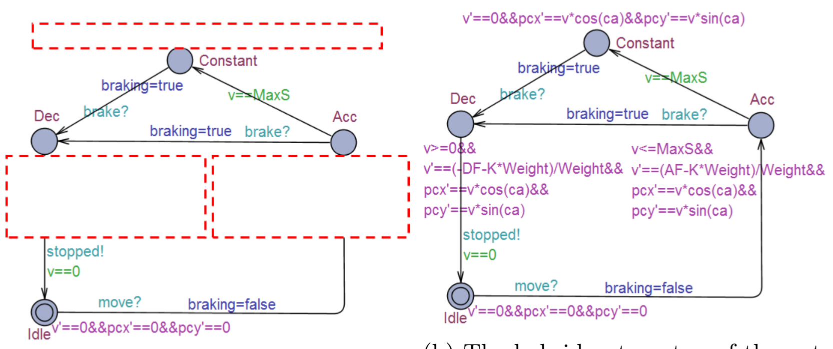
### 8.5.1 Patterns for the Execution Unit

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:

$$\dot{x}(t) = v(t)cos\theta(t) \quad \dot{y}(t) = v(t)sin\theta(t) \tag{8.2}$$

$$\dot{\theta}(t) = \omega(t), \tag{8.3}$$

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.



(a) The skeleton of the pattern

(b) The hybrid automaton of the pattern

Figure 8.4: The pattern of the linear motion component in the execution unit

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 8.4a, 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 8.2 should be encoded in the refinement of each location in the blank boxes in 8.4a. Figure 8.4b 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.

### 8.5.2 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 8.2) as an example. The process `ReadSensor` is firstly triggered

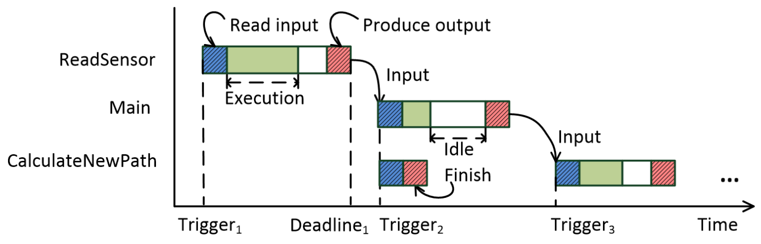


Figure 8.5: Process scheduling

at the moment $Trigger_1$ when the process reads data from sensors and runs its function as illustrated in Figure 8.5. 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 benefits 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

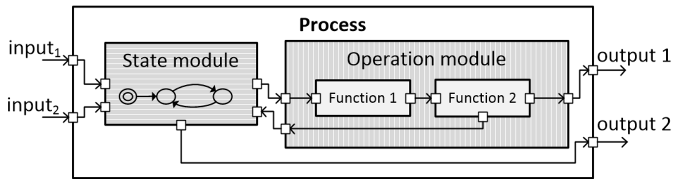


Figure 8.6: A process model example

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 8.6 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.

### 8.5.3 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 8.7, 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 8.8a. 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 8.5.2. The blank box indicates the process's own function that is created in an ad-hoc fashion, so it is not part

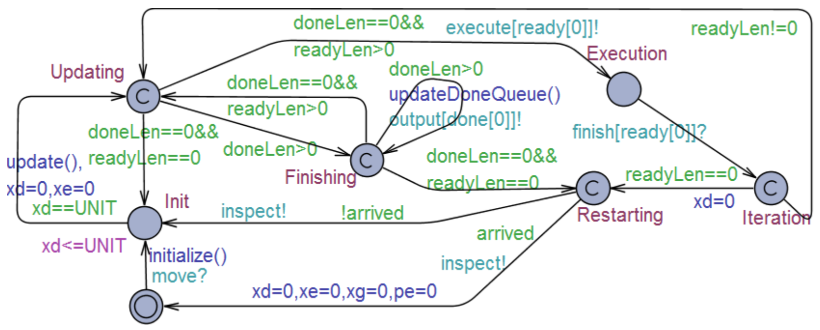
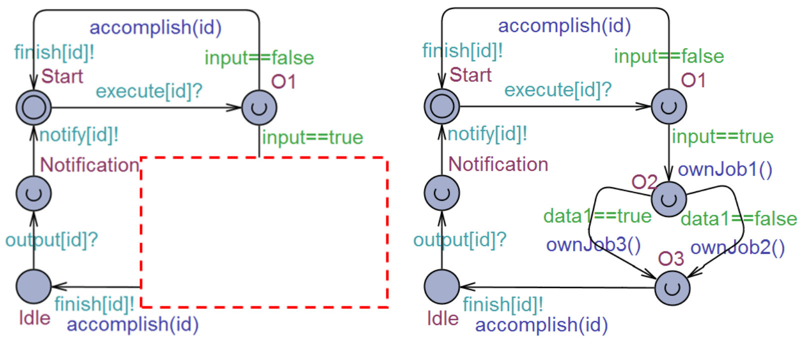


Figure 8.7: The pattern of the scheduler

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 8.8b. 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 8.6. 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.

## 8.6 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 8.8: The pattern of a generic process

### 8.6.1 Formal Model of the Control Unit

The control unit contains three parallel processes (Figure 8.2). `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 8.9 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.

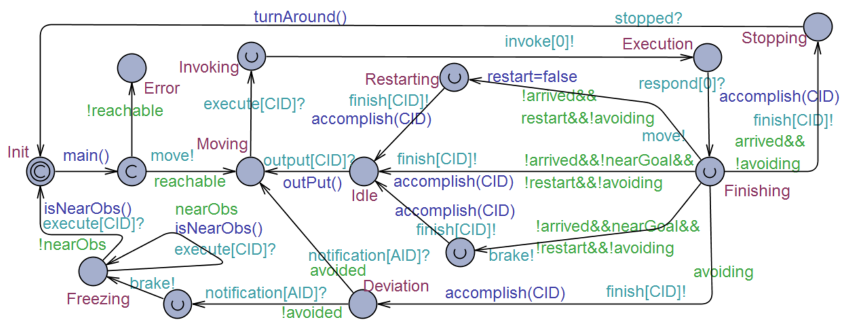


Figure 8.9: The automaton of the state module of the process `Main`

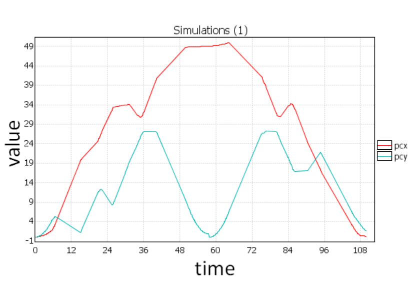
## 8.6.2 Statistical Model Checking of the AWL Formal Model

**Environment configuration**. In the following we consider a continuous map with the size $55 \times 55$, 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.

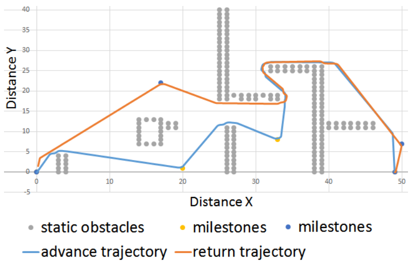
**Path generation and following**. Given a start and a goal and a set of milestones, the AWL must be able to calculate a safe path passing through them orderly avoiding static obstacles if the path exists and follow it. To verify this requirement, we first simulate the model in UPPAAL SMC using the command:

$$simulate\ 1[<= 110]\ \{pcx, pcy\} \tag{8.4}$$

where `pcx` and `pcy` are the real-valued coordinate of the AWL. Figure 8.10a 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 8.10b. The AWL perfectly follows the generated path that avoids all



(a) Coordinate changing of the AWL



(b) Moving trajectory of the AWL in Excel

Figure 8.10: Moving trajectory of the AWL generated by the command {`simulate 1[<=110] pcx,pcy`} in UPPAAL SMC and exported in Excel

the static obstacles. But the simulation only runs one possible execution trace of the AWL model. Hence, we further verify the model with a query:

$$Pr[<= 70](<>\ arrived\ \&\&\ counter <= 60) \tag{8.5}$$

$$Pr[<= 110]([]\ followedPath) \tag{8.6}$$

where `arrived` and `counter` in query 8.5 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 8.6 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:

$$simulate\ 1[<= 110]\ \{pcx, pcy, ocx[0], ocy[0], ocx[1], ocy[1], ocx[3], ocy[3]\} \tag{8.7}$$

$$Pr[<= 110]([]\ !collided) \tag{8.8}$$

Arrays `ocx` and `ocy` in query 8.7 represent the positions of moving obstacles at x and y axes. The trajectories got from query 8.7 is shown in Figure 8.11, 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

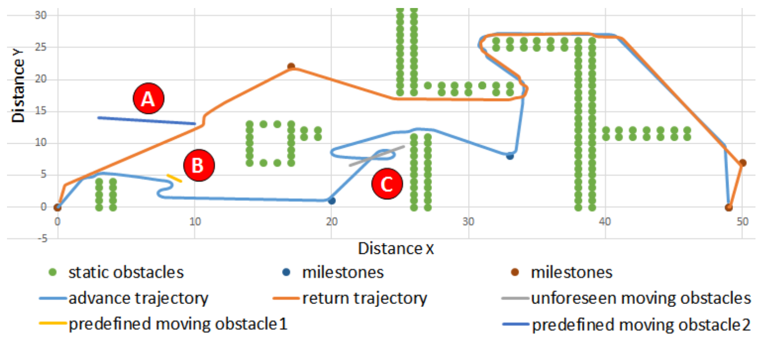


Figure 8.11: The trajectory of the AWL in a map with three moving obstacles

of two trajectories at “C” does 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 8.8 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.

## 8.7 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.

## 8.8 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.

## Acknowledgments

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 9

# Paper B: Verifiable Strategy Synthesis for Multiple Autonomous Agents: A Scalable Approach

Rong Gu, Peter G. Jensen, Danny B. Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist
Published in Journal of Software Tools for Technology Transfer, Springer, 2022.

**Abstract**

Path planning and task scheduling are two challenging problems in the design of multiple autonomous agents. Both problems can be solved by the use of exhaustive search techniques such as model checking and algorithmic game theory. However, model checking suffers from the infamous state-space explosion problem that makes it inefficient at solving the problems when the number of agents is large, which is often the case in realistic scenarios. In this paper, we propose a new version of our novel approach called MCRL that integrates model checking and reinforcement learning to alleviate this scalability limitation. We apply this new technique to synthesize path planning and task scheduling strategies for multiple autonomous agents. Our method is capable of handling a larger number of agents if compared to what is feasibly handled by the model-checking technique alone. Additionally, MCRL also guarantees the correctness of the synthesis results via post-verification. The method is implemented in UPPAAL STRATEGO and leverages our tool *MALTA* for model generation, such that one can use the method with less effort of model construction and higher efficiency of learning than those of the original MCRL. We demonstrate the feasibility of our approach on an industrial case study: an autonomous quarry, and discuss the strengths and weaknesses of the methods.

## 9.1 Introduction

With the rise of artificial intelligence (AI), autonomous agents such as driverless cars, drones, and autonomous construction equipment, are increasingly integrated in all aspects of society. Autonomy requires that the involved agents are able to sense the often unpredictable environment and act on changes over time in order to pursue their goals [1]. For instance, in a construction site, the autonomy of the agents (machines) bear the promise of increasing people's safety, while improving industrial productivity by automating repetitive tasks. Two major problems need to be solved to achieve the autonomy during operations: *path planning* and *task scheduling*. Computing both automatically is called *mission plan synthesis*. Path planning aims to calculate a path that visits all target positions (a.k.a. milestones) and avoids static obstacles. Algorithms like A* [2], Theta* [3], and Rapidly-exploring Random Tree [4] are adopted widely for calculating the shortest path between two points in a 2-D map. While path plans specify the movement between every two milestones, the order in which tasks should be completed at milestones is often dealt with as a subsequent optimization problem. The optimization problem of task scheduling is often paired with additional constraints, such as finishing tasks in a certain order, and repeating some tasks until the agents are informed to execute other tasks. The requirements on tasks can involve temporal conditions, e.g., "always start task $A$ before task $B$ is finished", and timing constraints, e.g., "always finish all tasks within 8 hours". All these constraints make the task scheduling difficult to complete in practice, in particular if constraints on computation time are given. In fact, a simplified version of task scheduling is the classic *job-shop* problem [5], which is NP hard [6].

In our previous work [7], we proposed an approach based on Timed Automata (TA) and Timed Computation Tree Logic (TCTL) to formally describe the agents' movement and task execution, as well as their requirements, respectively, to facilitate synthesis of plans by model checking. The approach has been implemented as a tool named TAMAA (Timed-Automata-based Mission planner for Autonomous Agents). The tool shows the feasibility of solving the mission-planning problem by using model checking when time of movement and task execution are fixed. However, TAMAA has two limitations: (i) if moving and executing tasks take unpredictable durations, TAMAA fails to generate complete mission plans that address all eventualities; (ii) TAMAA alone does not scale well with the number of agents growing, as the state space of the model explodes when the number of agents becomes large.

In this paper, we first solve problem (i) by synthesizing comprehensive strategies of timed games (TG), which use time intervals instead of fixed times as the moving and task execution times. TG are solvable by UPPAAL TIGA [8], which is for synthesizing strategies of TG. The TAMAA-generated TA can be re-used and easily converted into TG by labeling actions as controllable and uncontrollable ones in UPPAAL TIGA. As the TG models consider all possible times of task execution and moving within given intervals, and UPPAAL TIGA utilizes liveness properties to find the state-action pairs of the models that always eventually reach the goal states, the results represent the complete mission plans that address all eventualities.

However, as the synthesis in UPPAAL TIGA is still based on exhaustive symbolic exploration, this method inevitably suffers from the same state-space explosion problem as ordinary TAMAA. The state-space-explosion problem is one of the most stringent issues when employing exhaustive search techniques such as model checking [9], therefore many studies have explored ways of fighting it [10, 11]. To solve problem (ii), we proposed a novel method called *MCRL* [12] that combines model checking with reinforcement learning [13] to synthesize mission plans for large numbers of agents. Instead of exhaustively exploring the state space, MCRL samples the state space randomly within a time frame, and then uses these samples to train the agent models so that their behavior becomes increasingly efficient in reaching their goals such as finishing all tasks. Since the method does not need to traverse every state of the model, state-space explosion is avoided.

In this paper, we improve the original MCRL by integrating it with UPPAAL STRATEGO[1] [14], which is a tool that integrates the UPPAAL model checker, simulation, algorithmic synthesis (i.e., UPPAAL TIGA), and learning-based synthesis. Thanks to the integration, we can merge the sampling phase and the training phase of MCRL so that the temporary synthesis results can be used in the simulation and accelerate it to get to the goal state. Specifically, after each round of simulation, the sampled trace is provided to a learning module, which runs *Q-learning* [15] to populate a Q-table. Q-learning is a reinforcement learning algorithm that calculates a value for each state-action pair in the trace.

[1]https://people.cs.aau.dk/∼marius/stratego/

The state-action pairs and their values are stored in the Q-table, which is then used as a strategy. *Strategies* are mission plans that constantly provide suggestions of actions to the agents, at each of their states. The suggested actions include moving to a certain milestone, or executing a certain task. After the learning algorithm is invoked, an intermediary strategy is generated, which does not necessarily cover all the eventualities in the unpredictable environment. However, it is still input into the next round of simulation so that the simulator explores the state space in a heuristic way, by increasing the probabilities of choosing the actions that have higher values than other actions of the same state in the Q-table. In this way, the simulation can get to the goal state increasingly likely and faster.

Although exhaustive model checking suffers from state-space explosion, it is beneficial at ensuring the correctness of the synthesized strategies, that is, the latter satisfy all requirements, and the completeness, meaning that the synthesized strategies cover all the eventualities in the unpredictable environment. Therefore, we leverage exhaustive model checking after a strategy is synthesized by the Q-learning algorithm, to verify if the agent models behave according to the requirements specification, under the control of the strategy. In this work, we further extend the model checker of UPPAAL STRATEGO to support the exhaustive verification of the learned strategies. In this way, model checking and reinforcement learning are combined effectively by our method (MCRL), in solving the mission-plan synthesis problem of multiple autonomous agents. Moreover, the new version of MCRL reuses the automation of model construction provided by our toolset named *MALTA*.

To summarize, this paper is an extension of our previous work [12] and the new contributions are:

- An improved version of TAMAA that employs UPPAAL TIGA to synthesize mission plans (i.e., strategies) that consider all the eventualities in the respective environments.

- A new version of MCRL integrated in UPPAAL STRATEGO, which provides advantages such as a merged phase of sampling and training that benefit each other. The new version of MCRL is implemented in an extensible scheme with the help of UPPAAL STRATEGO, so that users can replace the learning algorithm with their own pre-compiled libraries.

- Experimental evaluation of the new methods by applying them on an industrial case study to demonstrate their merits and weaknesses.

The remainder of the paper is organized as follows. In Section 9.2, we introduce the preliminaries of this paper, that is, definitions of timed automata, (stochastic) timed games, (stochastic) strategies, and reinforcement learning. Section 9.3 describes the problem of strategy synthesis for multiple agents, as well as its challenges. Section 9.4 presents all the solutions and their application ranges. This section provides a general view of the methods and describes their differences. In Section 9.5, we overview our previous method TAMAA, which is the foundation of the new methods, and introduce the improved version of TAMAA in UPPAAL TIGA. Section 9.6 continues with the introduction of the learning-based method for strategy synthesis. It first analyzes the root of the scalability problem of TAMAA, after which it describes the new MCRL. Section 9.7 presents the implementation of MCRL and the integration with UPPAAL STRATEGO, as well as the automated model generation supported by our existing toolset. Next, we describe the evaluation experiments in Section 9.8, where we present the results of the experiments, as well as a discussion of the merits and weaknesses of the simulation-based methods and the improved version of TAMAA. In Section 9.9, we compare to related work, before concluding the paper in Section 9.10.

## 9.2 Preliminaries

### 9.2.1 Timed Automata and Timed Games

**Definition 1.** *A* Timed Automaton *TA [16] is a tuple:*

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

*where $L$ is a finite set of locations, $l_0$ is the initial location, $X$ is a finite set of non-negative real-valued clocks, $\Sigma$ is a finite set of actions, $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is a finite set of edges, where $\mathcal{B}(X)$ is the set of guards over $X$, that is, conjunctive formulas of clock constraints of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in X$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and $I : L \to \mathcal{B}(X)$ assigns an invariant to each location.* □

The semantics of a TA $\mathcal{A}$ is defined as a *timed transition system* over states $(l, v)$, where $l$ is a location and $v \in \mathbb{R}^X$ represents the valuation of the clocks on that location, with the initial state $s_0 = (l_0, v_0)$, where $v_0$ assigns all clocks in $X$ to zero. There are two kinds of transitions:

(i) *delay transitions*: $(l, v) \xrightarrow{d} (l, v \oplus d)$, where $v \oplus d$ is the result obtained by incrementing all clocks of the automaton with the delay amount $d$ such that $v \oplus d \models I(l)$, and

(ii) *discrete transitions*: $(l, v) \xrightarrow{a} (l', v')$, corresponding to traversing an edge $l \xrightarrow{g,a,r} l'$ for which the guard $g$ evaluates to *true* in the source state $(l, v)$, $a \in \Sigma$ is an action, $r$ is the clock reset set, and clock valuation $v'$ of the target state $(l', v')$ is obtained from $v$ by resetting all clocks in $r$ such that $v' \models I(l')$.

We denote the timed transition system of a TA $\mathcal{A}$ by $S_{\mathcal{A}}$. A run $\pi$ of a TA $\mathcal{A}$ is a sequence of alternating delay and discrete transitions of its $S_{\mathcal{A}}$: $\pi = (l_0, v_0) \xrightarrow{d_1} (l_0, v_1) \xrightarrow{a_1} (l_1, v_1') \xrightarrow{d_2} ... \xrightarrow{d_n} (l_{n-1}, v_n) \xrightarrow{a_n} (l_n, v_n')$, where $d_i$ refers to a delay transition and $a_i$ refers to a discrete transition. We denote the set of finite runs of $\mathcal{A}$ starting from $(l_0, v_0)$ as $\Pi_f(\mathcal{A})$.

A *Timed Game* $\mathcal{G}$ (TG) [17] is a TA whose actions $\Sigma$ are partitioned into controllable ($\Sigma_c$) and uncontrollable ($\Sigma_u$) actions. The timed transition system, runs, and a set of runs of a TG are denoted as $S_{\mathcal{G}}$, $\pi$, and $\Pi(\mathcal{G})$, respectively. TG is a useful mathematical model, suitable to describe a system consisting of several players that compete or collaborate to win the game, e.g., by finishing their tasks. Each player can take arbitrary numbers of actions before other players act. The numbers depend on the design of the TG. Informally, a strategy is a function that during the course of the TG constantly suggests the players what to do next in order to win the game. The suggestion is either a controllable action $a \in \Sigma_c$ or a delay. Delays in strategies are denoted as $\lambda$, which do not indicate the lengths of delays, whereas the symbol $d_i$ used in the definition of runs refers to concrete delays with specific lengths. The formal definition of strategies is as follows, where $last(\pi_f)$ is used to denote the last state of a finite run $\pi_f$:

**Definition 2** (Strategy)**.** *Let $\mathcal{G} =< L, l_0, X, \Sigma_c \cup \Sigma_u$, $I >$ be a TG. A strategy $\sigma$ over $\mathcal{G}$ is a partial function: $\pi_f \rightarrow \Sigma_c \cup \{\lambda\}$ such that for any finite run $\pi_f$ ending in state $q$ (i.e., $q = last(\pi_f)$), if $a \in \sigma(\pi_f) \cap \Sigma_c$, then there must exist a transition $q \xrightarrow{a} q' \in S_{\mathcal{G}}$.* □

Definition 2 indicates that a strategy is a function that takes finite runs of the TG as input and output controllable actions or delays as suggestions of actions to the agents. If the strategy $\sigma$ is *memoryless*, that is, the decisions on actions depend only on the current state, it can be represented as a function: $last(\pi_f) \rightarrow \Sigma_c \cup \{\lambda\}$. In this paper, we focus on *memoryless* and *non-lazy winning* strategies [18], which either urgently decide on a controllable action or *wait* until the environment acts[2].

### 9.2.2 Stochastic Timed Games and Stochastic Strategies

In principle, more information is often known of the environment, for instance, the likelihood of actions or the probability distribution of delays. In this section, we consider *Stochastic Timed Games*, where a stochastic environment is assumed. The environment makes choices of delay and uncontrollable actions stochastically, according to a density function for a given state. We define the Stochastic Timed Game as a Timed Markov Decision Process (TMDP) [18]:

**Definition 3** (Stochastic Timed Games). *A Stochastic Timed Game (STG) is a TMDP $\mathcal{P} =< \mathcal{G}, \mu^u >$, where $\mathcal{G} \; = < L, l_0, X, \Sigma_c \cup \Sigma_u, E, I >$ is a TG, and $\mu^u$ is a family of density-functions. Let $\mu^u_q(d,u) \in \mathbb{R}_{\geq 0}$ be a member of $\mu^u$, which assigns a probability density of the environment taking the uncontrollable action "u" after a delay of "d" at the state "q", where $\{\mu^u_q : \exists l \exists v.q = (l,v)\}$, $u \in \Sigma_u$ is an uncontrollable action, and $q$ is a state $(l,v)$.* □

Stochastic strategies [18] for STG are correspondingly defined as follows:

**Definition 4** (Stochastic Strategy). *A stochastic strategy $\mu^c$ for a STG is a family of density-functions. Let $\mu^c_q(d,c) \in \mathbb{R}_{\geq 0}$ be a member of $\mu^c$, which assigns a probability density of the controller taking the controllable action "c" after a delay of "d" from state "q", where $\{\mu^c_q : \exists l \exists v.q = (l,v)\}$, $c \in \Sigma_c$ is a controllable action, and $q$ is a state $(l,v)$.* □

**Remark 5.** *The STG models are for sampling the state-action pairs in the corresponding TG. They are used in the simulation and learning phases of MCRL. The TG models, which reflect the agents' behavior*

[2]This kind of strategies are shown to suffice for optimal scheduling of Duration Probabilistic Automata [19].

*more realistically, are used in the verification phase of MCRL, and the algorithmic synthesis in UPPAAL TIGA.*

### 9.2.3 UPPAAL, UPPAAL TIGA, and UPPAAL STRATEGO

#### UPPAAL

UPPAAL [20] is a state-of-the-art model checker for real-time systems. It supports modeling, simulation, and model checking, and uses an extension of TA as the modeling formalism. We use an example depicted

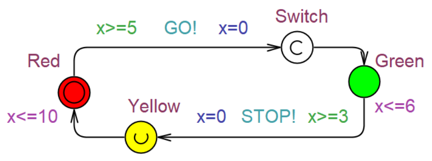


Figure 9.1: An example of a UPPAAL timed automaton (UTA) of a traffic light

in Fig. 9.1 to illustrate a simple UPPAAL TA (UTA) modeling traffic lights. *Locations* are circles, such as the ones labeled `Red` and `Green`, which model the two colors of the traffic lights. The initial location is the double circle (i.e., `Red`). One UTA can have only one initial location. The UTA's *edges* are directed lines that connect locations, which can be decorated by *guards*. A clock variable `x` is defined to measure the elapse of time, and used in the *invariants* on locations (e.g., `x<=6`), which specify how long the UTA can delay on that location, and *guards* on edges (e.g., `x>=3`).

A *network* of UTA models a parallel composition of UTA that can synchronize via *channels* (i.e., $a!$ is synchronized with $a?$ by handshake). In Fig. 9.1, the edges are labeled with channels named `STOP` and `GO`, which synchronize this UTA with other UTA. In UPPAAL, there are two special kinds of locations, namely *urgent* and *committed* locations. *Urgent* locations are denoted by encircled u, and require that the time does not elapse on those locations (e.g., `Yellow`); *committed* locations are denoted by encircled c, and require that not only no time elapses there but also the next edge to be traversed must start from one of the committed locations in the network of UTA (e.g., `Switch`). UTA also extends TA by introducing discrete data variables that can be updated via functions on edges. Functions are written in a subset of the C language. Clocks can be reset over edges, e.g., `x=0` in Fig. 9.1.

The UPPAAL queries that we verify in this paper are properties of the following form, where $p$ is an atomic proposition over the locations, clocks, and data variables of the UTA: (i) **Invariance**: $A$`[]`$p$ meaning that for all runs, for all states in each run, $p$ is satisfied, (ii) **Liveness**: $A$`<>`$p$ meaning that for all runs, $p$ is satisfied by at least one state in each run, and (iii) **Reachability**: $E$`<>`$p$ meaning that there exists a run where $p$ is satisfied by at least one state of the run.

#### UPPAAL TIGA and UPPAAL STRATEGO

UPPAAL TIGA [8] is an extension of UPPAAL, which supports solving games based on TG with respect to the temporal properties aforementioned. In this paper, we use UPPAAL TIGA to solve our task scheduling problem in the first solution based on game-theoretic synthesis. UPPAAL STRATEGO [14] is a tool that integrates UPPAAL with two of its branches, that is, UPPAAL SMC [21] (statistical model checking) and UPPAAL TIGA [8]. In addition, it also supports learning-based algorithms for solving STG, and we use this tool to develop our second solution to strategy synthesis that is based on simulation and learning.

### 9.2.4 Reinforcement Learning

MCRL employs *reinforcement learning* (*RL*) for strategy synthesis. *RL* is a kind of machine learning method for training reactive systems by rewarding desired behaviors and/or punishing undesired ones. Agents that constantly act in an environment and receive feedback (i.e., rewards/penalties) from the environment are reactive systems. *RL* aims to calculate how agents should take actions in an environment, in order to maximize the accumulated rewards of actions. Model-free *RL*, such as *Actor-Critic algorithms* [22], relies on samples from the environment, which can be a model or a real environment, to estimate the rewards of the next state-action pairs. Model-based *RL*, such as *Dynamic Programming* [13], uses the model's predictions or distributions of the next state-action pairs and their rewards to calculate optimal actions.

*Q-learning* is one of the model-free algorithms, which, at the limit, converges to *optimal policies* for reactive agents in a stochastic environment. Policies are associated with a state-action value function called *Q function*. The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \underset{a'}{MAX}\; q^*(s',a')], \tag{9.2}$$

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 $s$, $\gamma$ is a discounting value, $s'$ is the new state coming from state $s$ by taking action $a$, and $\underset{a'}{MAX}\; q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair $(s',a')$. The Bellman equation calculates the rewards of state-action pairs by considering both the current reward and the discounted maximum future reward. Watkins [15] shows that under the assumption of sufficient repeated sampling, the Q-learning algorithm converges towards the optimal Q-values and thus the solution to the Bellman equations. These values are stored in Q-tables, which serve as the strategies that we aim to synthesize.

## 9.3 Problem Description and Analysis

In this section, we introduce the *autonomous quarry* that serves as the industrial case study provided by VOLVO Construction Equipment (CE) in Sweden. Based on this practical case study, we formulate our research problem and two associated challenges.

### 9.3.1 An Industrial Case Study: The Autonomous Quarry

As depicted in Fig. 9.2, the quarry contains various autonomous vehicles, e.g., trucks and wheel loaders, which are the agents in the environment.

A typical mission of the agents is to transport stones from stone piles to crushers. Specifically, wheel loaders first dig stones at the stones piles and load them into trucks that are responsible for transporting the stones to crushers. The primary crushers crush the stones into fractions, after which trucks load the crushed stones and transport the material to the secondary crushers, which is the final destination of the stones. During the transportation, the agents (that is, autonomous wheel loaders

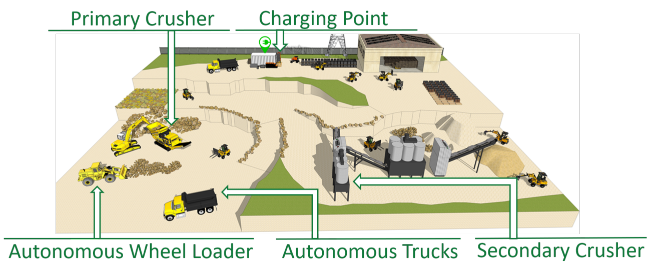


Figure 9.2: An example of an autonomous quarry

and trucks) must avoid static obstacles (e.g, holes and rocks on the ground, larger than given sizes) and dynamic obstacles (e.g., humans, other mobile machines). In brief, these agents must be able to plan their paths to the target positions (a.k.a. milestones) and schedule their tasks so that the entire mission could be accomplished respecting some requirements, e.g., quarrying $1500m^3$ of stones per day.

Generalizing from this case study, our research problem of mission planning involves task scheduling, path planning and following, and collision avoidance for multiple agents. In our previous work [23][24], we have introduced a solution for the collision-avoidance problem of dynamic obstacles, and proposed a method for verifying this function. In this paper, we focus on synthesizing static mission plans, while assuming that the dynamic avoidance among agents as well as other moving obstacles functions correctly.

## 9.3.2 Problem Analysis

Algorithms such as Theta* [3] and RRT [4] are capable of computing collision-free paths between two milestones. We adopt the Theta* algorithm to solve path-planning in this study and the experiments, as the algorithm is especially good at generating smooth paths with any-angle turning points, in 2-D maps. Note that, our toolset (introduced in Section 9.7) supports more path-planning algorithms. After the paths are calculated, the execution order of tasks on milestones must be decided to achieve correct and efficient strategies. Based on the requirements from VOLVO CE, we formulate and categorize the requirements of tasks as follows:

- *Milestone Matching.* Tasks must be performed at the right milestones, e.g., digging stones must be carried out at stone piles.
- *Task Sequencing.* The task execution order must be correct, e.g., unloading stones into the primary crusher must be executed after digging stones is finished, but before loading stones starts.
- *Timing.* All tasks that contribute to the goal (e.g. transporting 10 tons of stones to the secondary crusher) must be finished within a prescribed time (e.g. within 1 hour).

*Task scheduling* now reduces to synthesizing a plan of task execution such that, by following the plan, agents can work independently or collectively to accomplish the entire mission according to the requirements. The classic scheduling problem called the *job-shop* problem [5] is a simplified version of the task scheduling. Being an NP-hard problem, even a simple instance of the job-shop problem with very restrictive constraints remains difficult to solve [6]. Additionally, our task-scheduling problem poses some unique extra challenges, as described in the next section. For simplicity, henceforth, we call the problem of path planning and task scheduling for autonomous agents as *mission planning.*

### 9.3.3 Non-determinism and Scalability of Mission Planning

Different from the classic job-shop problem, there are two types of uncertainties existing in the environment that must be considered in the *mission-planning* phase, that is, the *non-deterministic execution time* of tasks and *non-deterministic duration* of agent movement.

- *Non-deterministic task execution time.* The execution time of a task is usually a time interval between the BCET (best-case execution time) and WCET (worst-case execution time) of the respective task.
- *Non-deterministic movement time.* The devices at some milestones could be exclusively occupied by agents. Therefore, other agents that are approaching these milestones must wait until those devices are released, respectively, and then start their tasks. This yields a non-deterministic movement time.

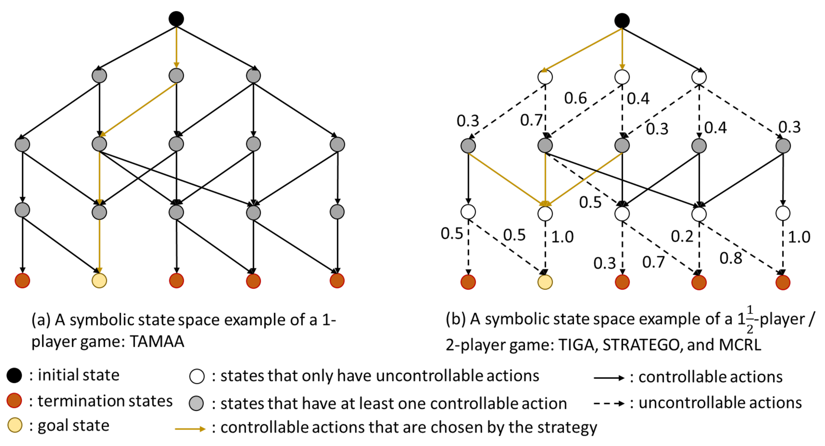

0.3
0.7
0.6
0.4
0.3
0.4
0.3
0.5
0.5
0.5
1.0
0.3
0.7
0.2
0.8
1.0
(a) A symbolic state space example of a 1-player game: TAMAA
(b) A symbolic state space example of a $1\frac{1}{2}$-player / 2-player game: TIGA, STRATEGO, and MCRL
: initial state
: termination states
: goal state
: states that only have uncontrollable actions
: states that have at least one controllable action
: controllable actions that are chosen by the strategy
: controllable actions
: uncontrollable actions

Figure 9.3: Examples of symbolic state spaces of different models of games. Probabilities in (b) are only used in $1\frac{1}{2}$-player games

These features complicate our task scheduling even more than in the classic job-shop case. Our target is not only calculating mission plans, but also guaranteeing their correctness, that is, showing that the synthesized mission plan (a.k.a., strategy) satisfies all the requirements, and that it is complete, namely, covers all eventualities in the environment.

In our previous work [7], we have proposed an approach called TAMAA, based on the model-checking technique, to synthesize mission plans for agents. This approach can automatically generate mission plans, assuming feasible numbers of milestones and tasks up to 100. However, the approach cannot cover all eventualities when the environment is non-deterministic. Additionally, when the number of agents exceeds 5, TAMAA exhausts the physical memory due to the notorious state-space-explosion problem of model checking [9].

## 9.4 Overall Description of the Solutions

Facing the limitation of TAMAA, we propose two solutions in this paper, that is, 1) a game-theoretic synthesis (i.e., TIGA), and 2) a simulation-based synthesis (i.e., MCRL). Table 9.1 lists the solutions and their

Table 9.1: Summary of all solutions.

| | TAMAA | TIGA | MCRL | STRATEGO |
|---|---|---|---|---|
| Model | UTA | TG | TG & STG | STG |
| Game | 1 player | 2 player | 2 player | $1\frac{1}{2}$ player |
| Techniques | Model Checking [7] | Symbolic On-The-Fly Algorithm [17] | Reinforcement Learning & Model Checking [12] | Reinforcement Learning [14][25] |

characteristics. TAMAA uses UTA as the modeling language and is suitable for 1-player games, in which agents have the full control over their environment. As depicted in Fig. 9.3(a), the agent models in TAMAA have no uncontrollable actions, which means the agents can totally control their movement and task execution times. Therefore, the goal of TAMAA is to find the best mission plans that finish all tasks the fastest.

However, strategies of $1\frac{1}{2}$-player and 2-player games can only choose controllable actions, whereas the uncontrollable actions are taken by the environment either non-deterministically in 2-player games, or stochastically in $1\frac{1}{2}$-player games (Fig. 9.3(b)). Therefore, the goal of 2-player games, solvable by UPPAAL TIGA and MCRL, is to find the comprehensive strategies that enable the agents to finish their tasks no matter which and when uncontrollable actions are taken. Taking into account the probabilities of performing the uncontrollable actions, the goal of $1\frac{1}{2}$-player games, solvable by UPPAAL STRATEGO, is to find the strategies that have the highest probability of finishing all tasks.

In summary, different methods are suitable for different applications, and have their own advantages and disadvantages. When stochastic behaviors are observed in the system, $1\frac{1}{2}$-player games and UPPAAL STRATEGO can provide a suitable solution. When agents can fully control their task execution times, 1-player games and TAMAA can be the right choice (Section 9.5). When the task execution times are flexible rather than fixed and the uncontrollable actions are non-deterministic, UPPAAL TIGA (Section 9.5) and MCRL (Section 9.6) are capable of handling the problem.

UPPAAL TIGA is sound and complete in the sense that when a strategy is synthesized, it is guaranteed to be correct by construction, and conversely, when such a strategy exists in the state space of the model, UPPAAL TIGA is able to find it. However, UPPAAL TIGA suffers from the scalability problem as the method relies on the exhaustive graphic search. MCRL uses a simulation-based method for synthesis and proposes a post-verification of the synthesized strategies, which alleviates the scalability problem while sacrificing the completeness of the method, that is, although MCRL has the ability to deal with more agents than UPPAAL TIGA, it does not guarantee to synthesize a strategy even if such strategy exists.

We will introduce these methods in detail in Sections 9.5 and 9.6, and then compare their performance in different application scenarios in Section 9.8.

## 9.5 Solution 1: Game-Theoretic Synthesis

In this section, we introduce the first solution, that is, our *game-theoretic synthesis*, which is based on an exhaustive search of the state spaces of agent models. We have two methods belonging to such kind of synthesis, namely the original TAMAA [7] and TAMAA in UPPAAL TIGA [8]. As aforementioned, the original TAMAA is designed to solve 1-player games, whereas TAMAA in UPPAAL TIGA leverage the models of TAMAA and the algorithmic method of synthesis of UPPAAL TIGA to synthesize complete plans that take into account any (possibly antagonistic) environmental action. First, we overview TAMAA, which provides an automatic model generation and synthesis of mission plans for 1-player games.

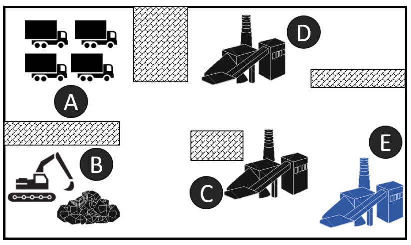
### 9.5.1 Overall Description of TAMAA

TAMAA [7] enables users to configure their agents, tasks, and working environment in a graphical user interface (GUI), and automatically generate UTA networks that model the movement and task execution of agents. After users finish the configuration, UPPAAL is called to verify the UTA models in order to generate runs that satisfy various properties. The verification used in TAMAA is not for checking if the model is correct or not, but for generating runs of the model, which are then parsed to generate mission plans.

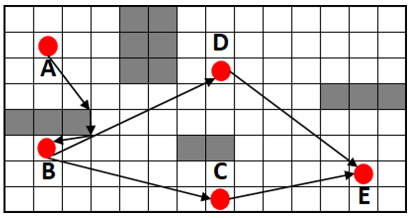
### 9.5.2 Mission-Plan Synthesis by TAMAA

To pave the foundation of the synthesis method, we first elaborate the UTA models generated by TAMAA by an example illustrated in Figure

9.4a. These models are also used in the improved solution of TAMAA with some slight adjustments (see Section 9.5.3).



(a) An example of quarries

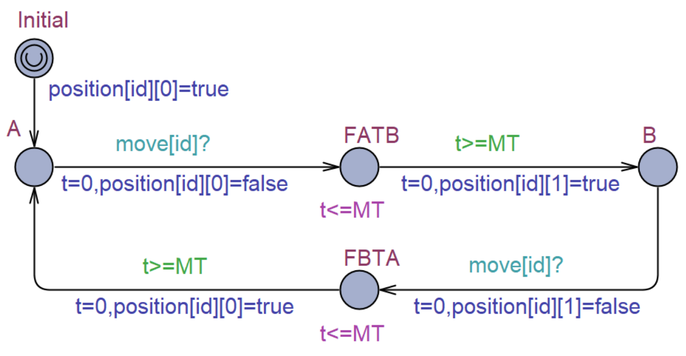


(b) Map decomposition and the calculated paths

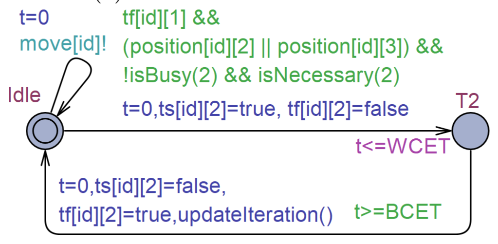
Figure 9.4: An example of calculating paths by decomposing the map and running the Theta* algorithm.

In the quarry example, there are four autonomous trucks starting at milestone $A$, which aim to transport stones from milestone $B$ to the primary crusher at milestone $C$ or $D$, and eventually go to the secondary crusher at milestone $E$. A wheel loader is working at milestone $C$ to dig stones and load them into the trucks. Only the autonomous trucks are the agents that we are interested in. First, the environment is decomposed into a Cartesian grid and the Theta* algorithm [3] is executed to calculate the shortest paths among milestones $A$ - $E$ (See Figure 9.4b). Note that the trucks only need to choose one primary crusher at position $C$ or $D$, to unload stones.

Next, UTA models are automatically generated by TAMAA, based on the shortest paths. For brevity, in Figure 9.5a, we show a part of the UTA model in UPPAAL describing the movement of the autonomous trucks between milestones $A$ and $B$. The movement to other milestones is modeled in a similar way. *Locations* `A` and `B` represent milestones $A$ and $B$, respectively. The outgoing edge from the urgent initial location to location `A` indicates that the trucks start from milestone $A$. *Locations* `FATB` and `FBTA` are created to count the traveling time between `A` and `B`. A constant variable `MT` stores the traveling time. The agent is only allowed to move when it is not executing any tasks. Therefore, *channel* `move[id]` is used to synchronize the transitions in the movement UTA with the task execution UTA (Figure 9.5b) so that the moving actions are only enabled when the agent is idle, where the variable `id` refers to



(a) Part of a movement UTA


t=0
move[id]!
tf[id][1] &&
(position[id][2] || position[id][3]) &&
!isBusy(2) && isNecessary(2)
Idle
t=0,ts[id][2]=true, tf[id][2]=false
T2
t<=WCET
t=0,ts[id][2]=false,
tf[id][2]=true,updateIteration()
t>=BCET

(b) Part of a task execution UTA

Figure 9.5: UTA models of an agent's movement and task execution

the current agent in both the movement UTA and the task execution UTA. A two-dimensional Boolean array named `position` is updated in this UTA, in which each element stores whether a certain milestone is being occupied by an agent. To model the agents' movement on the paths in Fig. 9.4b, TAMAA instantiates movement UTA similar to Fig. 9.5a.

The task execution UTA models the actions that an agent can choose to execute at a milestone. One such UTA is partly depicted in Fig. 9.5b, where location `Idle` represents the status of "no operation", when the agent is allowed to move, and location `T2` represents the task of unloading stones into a primary crusher. The self loop of location `Idle` labeled by *channel* `move[id]` regulates the movement UTA to start to move only when the task execution UTA is at location `Idle`. Two Boolean arrays named `ts` and `tf` are updated in the UTA, representing whether a task has been started or finished, respectively. Assuming trucks need to iterate their tasks multiple times before transporting all the stones, the guard of the incoming edge of location `T2` enables this edge if the following conditions are *true*: (i) the task of loading stones from the wheel loader is done (i.e., `tf[id][1]` is *true*), (ii) the agent must be at milestone $C$ or $D$, where the primary crushers are located (i.e., `position[id][2]` || `position[id][3]` is *true*), (iii) no other agent is executing this task (i.e., `!isBusy(2)` is *true*), and (iv) the task has not been done in this round of transportation (i.e., `isNecessary(2)` is *true*). Location `T2` has an invariant indicating that the execution time of the task must not exceed its WCET. Similarly, the guard on the outgoing edge of location `T2` ensures that task is finished after the execution time is greater than or equal to BCET. The function `updateIteration()` updates the integer of task iteration.

After the resulting UTA model is automatically generated by TAMAA, properties that formalize the requirements mentioned in Section 9.3.2 are also generated by using the configuration information and well-designed TCTL templates. The *Timing* requirement is used for synthesizing mission plans that finish all tasks within a prescribed time limit. Others are for verifying if the models guarantee that the mission plans are functionally correct. For brevity, we only show the TCTL property of the *Timing* requirement used in UPPAAL. The rest of the properties are reported in our previous work [7]. The TCTL reachability Query (9.3) checks if agents can accomplish their missions within $TL$ time units, where `ite` is an integer array storing the iteration of the tasks, that is, finishing all tasks once counts for one round, `x` is a clock variable that

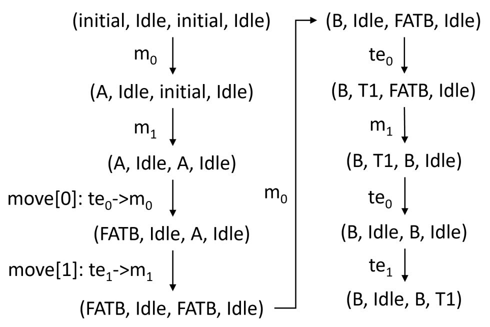


Figure 9.6: A segment of a run generated by TAMAA

is never reset, `N` and `M` are two integers indicating the number of agents and the requested iterations of tasks, respectively:

$$\texttt{E<> ((forall(i:int[0,N-1]) ite[i]>=M) \&\& x}\leq\texttt{TL)} \tag{9.3}$$

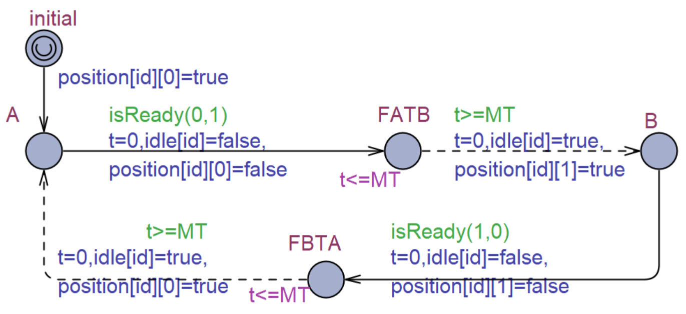
The target of mission planning in a 1-player game is to find the run that reaches the goal state where, for example, agents finish all the tasks. Fig. 9.6 depicts a segment of such run belonging to a model of 2 agents, where states of the models are symbolically represented by the locations of the UTA, $\texttt{m}_i$ and $\texttt{te}_i$ stand for actions in movement and task execution UTA of agent $i$, respectively, and $\texttt{move[i]:te}_i\texttt{->m}_i$ stands for the synchronized actions of starting to move. As depicted, all the actions are controllable by the agents (i.e., solid lines), which consecutively or alternately move the respective agent and execute tasks. They can stay at the same milestone (e.g., `B`) but cannot execute the same task (e.g., two `T1` cannot appear at the same state) unless the agents are not mutual exclusive of the task.

As explained in Section 9.4, runs like the one in Fig. 9.6 are mission plans of 1-player games. To obtain such runs, TAMAA uses the model checker of UPPAAL to verify the UTA models of agents against reachability properties in the form of Query (9.3). If the properties are satisfied, UPPAAL can generate runs that can be either the fastest, shortest, or random run, respectively. Hence, TAMAA can generate these three kinds of mission plans.

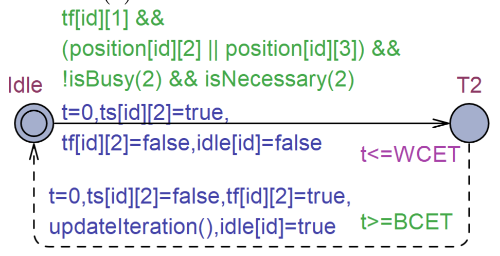
However, when the problem becomes a $1\frac{1}{2}$-player game or a 2-player game, TAMAA is not able to solve it, because the task execution times are decided by the uncontrollable actions taken by the environment. We need another method to deal with these problems such that the mission plans can cover all possible scenarios, even in the face of an antagonistic environment.

### 9.5.3 Synthesizing Strategies in UPPAAL TIGA

In this subsection, we apply UPPAAL TIGA instead of UPPAAL to synthesize strategies defined by Definition 2, which serve as the complete mission plans that the original TAMAA is not able to synthesize.



(a) Part of a movement TA


tf[id][1] &&
(position[id][2] || position[id][3]) &&
!isBusy(2) && isNecessary(2)
Idle
T2
t=0,ts[id][2]=true,
tf[id][2]=false,idle[id]=false
t<=WCET
t=0,ts[id][2]=false,tf[id][2]=true,
updateIteration(),idle[id]=true
t>=BCET

(b) Part of a task execution TA

Figure 9.7: TG models of an agent's movement and task execution in UPPAAL TIGA

We recast the models of TAMAA from the UTA formalism into the TG formalism of UPPAAL TIGA as follows. As depicted in Fig. 9.7a, the edge from location `FATB` (resp., `FBTA`) to location `B` (resp., `A`) is marked as uncontrollable. This change indicates that the decision of choosing a milestone to visit is made by the agent, whereas the duration of the movement to reach the milestone is determined by the environment. Note that the invariant on `FATB` (i.e., `t<=MT`) and the guard on the outgoing edge of `FATB` (i.e., `t>=MT`) force the duration to be `MT`. Similarly, in Fig. 9.7b, the edge from location `T2` to location `Idle` is marked as uncontrollable, indicating that the duration of the task is determined by the environment.

Besides the change of uncontrollable actions, the synchronization between the task execution UTA and movement UTA is removed to avoid the input non-determinism of random simulation in UPPAAL. Instead, a global Boolean array `idle` is introduced in the TG models to store whether the agents are idle or not. This array is used in the function named `isReady` in the movement TG, which returns *true* when the corresponding element in `idle` is *true* and the agent has not finished its requested iteration of tasks.

Query (9.3) is also adjusted to synthesize complete strategies that deal with the non-determinism of the environment, as follows:

$$\begin{aligned}\texttt{strategy st = control: A<> ((forall(i:int[0,N-1])}\\ \texttt{ite[i]} \geq \texttt{M) \&\& x}\leq\texttt{TL)}\end{aligned} \tag{9.4}$$

Query (9.4) applies the universal quantifier `A` on runs and the "eventually" temporal operator `<>` on states, which means that the synthesized strategy `st` must always guide the agents to finish their tasks for `M` rounds within `TL` time limit, no matter how long the time of task execution is.

As depicted by Fig. 9.8, the runs generated by UPPAAL TIGA contain controllable (solid lines) and uncontrollable actions (dashed lines). The other notions of the figure are the same as in Fig. 9.6. The first four steps in Fig. 9.6 and Fig. 9.8 are the same, being all controllable actions. The fifth step in Fig. 9.8 starts to be different, because it is an uncontrollable action, which means that the environment decides which actions to perform, instead of the agents. Assuming that the agents travel at the same speed, at state `(FATB, Idle, FATB, Idle)`, the environment can choose agent 0 to arrive at milestone $B$ first via the uncontrollable action in $\mathtt{m}_0$; or choose agent 1 to arrive first via the uncontrollable action in $\mathtt{m}_1$. The actions of finishing tasks are also uncontrollable, so the task execution times are uncertain from the agents' point of view. The strategies synthesized by UPPAAL TIGA are complete in the sense that no matter which and when uncontrollable actions are taken, the agents can always

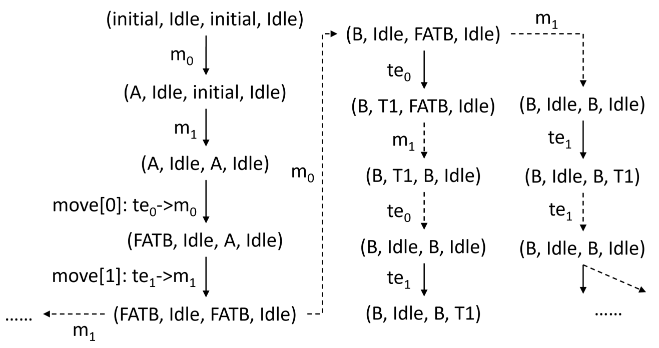


Figure 9.8: A segment of a strategy generated by UPPAAL TIGA

finish their tasks with respect to various requirements by following the strategies.

Although the strategies are now complete, since UPPAAL TIGA is also (in the worst case) exhaustively exploring the state space to synthesize strategies, the scalability problem of TAMAA still exists in UPPAAL TIGA. As depicted in Table 9.2, the number of explored states, and the computation time increase exponentially with the agent number growing linearly, which implies that UPPAAL TIGA encounters the state space explosion.

Table 9.2: Performance evaluation of synthesis in UPPAAL TIGA with different number of agents running 3 tasks among 3 milestones.

| Number of agents | Number of explored states | Computation time |
|---|---|---|
| 2 | 775 | 5 ms |
| 3 | 222,88 | 220 ms |
| 4 | 764,001 | 18.1 s |
| 5 | 33,312,229 | 53.8 mins |
| 6 | Out of memory | Unknown |

## 9.6 Solution 2: Simulation-Based Synthesis

In this section, we introduce our second solution for the task-scheduling problem, which is based on simulation and learning. First, we describe the root of the state-space-explosion problem that the both the original and improved versions of TAMAA have.

### 9.6.1 State-Space Exploration of TAMAA

The states of the agent models are the Cartesian product of states in each individual agent. Therefore, the state space of multiple agents grows exponentially with the number of agents growing linearly. The interleaving actions among the agents also increase the state space. Running TAMAA in UPPAAL TIGA requires searching the state space of the model. The essence of the method is about searching and storing the state space in order to find the runs that reach (respectively, avoid) certain states for reachability properties (respectively, safety properties). Since it relies on an implementation of an on-the-fly symbolic algorithm, UPPAAL TIGA may terminate before having explored the entire state space, which alleviates the state-space-explosion problem. However, the searching algorithm is either breadth-first, depth-first, or random, which is not heuristic because it constantly follows the same order of searching without using the historical information of the searched state space. Therefore, the synthesis method in UPPAAL TIGA can take a long time to find the desired runs when the state space is large. The simulation-based synthesis, which is presented in the next subsection, improves the method in this aspect.

### 9.6.2 Learning Strategies

Instead of using a symbolic and potentially exhaustive method, we study here the use of simulation-based synthesis algorithms such as *Q-learning* [15]. Rather than exploring the state space exhaustively, simulation-based methods sample the state space strategically, which happens often in a reactive manner, hence they can avoid state-space explosion. Nonetheless, simulation-based approaches sacrifice completeness over speed of synthesis, but gain the ability to accommodate a stochastic resolution of environment choices.

In this subsection, we go through the workflow of the new version

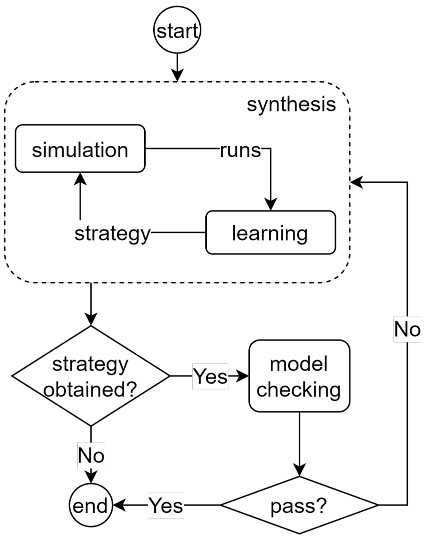


Figure 9.9: Workflow of MCRL

of MCRL, which is integrated with UPPAAL STRATEGO. In the rest of Section 9.6, we introduce the new features of the new MCRL while briefly introducing the functions and parameters in UPPAAL STRATEGO. For technical details of UPPAAL STRATEGO, readers are referred to the literature [14] [25].

As depicted in Fig. 9.9, MCRL explores the state space of the TG model via random simulation at the initial step, during which runs of the model are sampled. These runs serve as input to the learning algorithm to compute the rewards or penalties of the state-action pairs. As a result, the pairs belonging to the runs that reach the states where tasks are finished faster than those in other runs are assigned with higher rewards, whereas the pairs that end up into deadlocked states, or are wondering meaninglessly, are assigned with lower rewards or even penalties. The accumulated values (i.e., rewards or penalties) of the state-action pairs contribute to synthesize an intermediary strategy, which is then used in the next round of simulation until a user-defined number of runs is sampled. Specifically, the simulator exploits the intermediary strategy in its following rounds of simulation by increasing (respectively, decreasing) the probabilities of choosing the actions with higher values (respectively, lower values). In this way, the simulator can reach the goal state faster and easier than the previous rounds of simulation do. This integration of simulation and learning is not provided by the initial version of MCRL [12].

When a user-defined number of runs is sampled, a strategy is considered to be produced. The simulation-based synthesis cannot guarantee the correctness of the strategies. Therefore, we propose a post-verification of the strategies by using model checking. Specifically, the TG models are verified together with the synthesized strategies. When the model checker encounters multiple controllable actions during verification, it enquires the strategy to choose the ones with the highest values. Details of the verification are presented in Subsection 23. Strategies that pass the verification are guaranteed to be correct in the sense that they satisfy the temporal constraints of requirements. If the verification fails, a new iteration of the synthesis and verification can be carried out, where the user-defined number of runs is increased for a more thorough learning. In addition, the state space of the model is restricted by the synthesized strategy, which enables MCRL to deal with more complicated problems than TAMAA and UPPAAL TIGA do.

In the following subsections, we introduce the key definitions, algorithms, and techniques that are used in the new version of MCRL. Since UPPAAL STRATEGO integrates UPPAAL, UPPAAL SMC, and UPPAAL TIGA, the following algorithms and techniques are designed and implemented collectively in UPPAAL STRATEGO.

### Model Conversion

The initial step of MCRL is random simulation (see Fig. 9.9). The non-deterministic choices of actions in the synthesis of UPPAAL TIGA are replaced by random sampling of actions in the simulation. Consequently, the TG must be converted into STG by assigning probabilities to actions.

Fig. 9.10 shows the model conversion in the process of learning and verifying a strategy. In step 1, the task-execution TG is changed to an STG, where the probability of finishing a task between its BCET and WCET is uniformly distributed. Note that the uniform distribution can be changed to a user-defined distribution that can make the agents finish their tasks easier, in the simulation. However, it does not change the fact that the synthesized strategies lack a correctness guarantee, which

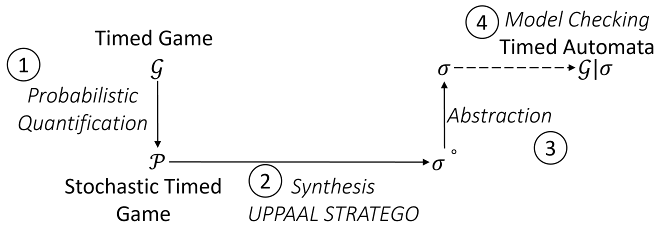


Figure 9.10: Overview of various models and strategies and their relations in UPPAAL TIGA and UPPAAL STRATEGO (adapted from [14])

means that the post-verification is needed anyway. The reason why we choose to use the uniform distribution is because it is the default distribution on time-bounded delays in UPPAAL STRATEGO[3]. Therefore, the syntactic structure of the TG does not need to be changed. We name the first step *probabilistic quantification* because it assigns quantitative probabilities to the actions that are originally chosen non-deterministically in UPPAAL TIGA.

Step 2 is MCRL's synthesis (also seen in Fig. 9.9), which learns a stochastic strategy $\sigma^\circ$ based on the STG. $\sigma^\circ$ is then abstracted to a strategy that does not contain any probability, in step 3. The abstraction of stochastic strategies is introduced in Subsection 23. In the final step, the TG and the synthesized strategy $\sigma$ are verified together by the model checker of UPPAAL STRATEGO[4]. This is supported by queries in the form of Query (9.7) that is introduced in Subsection 23.

The model conversion does not spoil our assumption of the environment, because the probabilities assigned to the uncontrollable actions in the TG are only used in the learning phase. The formal verification of the synthesized strategy is still by exhaustive model checking, which guarantees that the agents satisfy the requirements regardless of how the environment behaves.

[3]The uniform distribution is used in UPPAAL SMC by default. UPPAAL STRATEGO includes UPPAAL SMC.

[4]The model checker is UPPAAL [20], which is included in UPPAAL STRATEGO.

### Q-learning Algorithm

Although we adopt Q-learning [15] in this work, our framework is open for extension with any other learning algorithms. In order to apply Q-learning on our STG models of the agents, we first define the states and actions of the Q-table generated by the learning algorithm. To differentiate the states of STG, we define Q-states and Q-actions as follows.

**Definition 5** (Q-State)**.** *A Q-state is defined as the following tuple:*

$$\mathcal{QS} =< RT, CT, CP, ST >,$$

*where:*

- $RT \in \mathbb{N}^d$ *is a set of natural numbers denoting the iteration of executing all tasks for each agent, where d is the number of agents,*
- $CT \in \mathbb{N}$ *denotes the index of the current task,*
- $CP \in \mathbb{N}$ *denotes the index of the current milestone,*
- $ST$ *is a set of Boolean variables encoding the respective execution statuses of tasks (EST) of all agents.* □

**Definition 6** (Q-Action)**.** *A Q-action is defined as the following tuple:*

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

*where:*

- $MT \in \{1, 2\}$ *denotes the type of motion, i.e.,* $1$ : *moving,* $2$ : *executing a task, and*
- $TT \in \mathbb{N}$ *denotes the target of the motion, which can be a milestone or a task.* □

In practice, "$RT$" is declared as an array of integers in our UTA models. "$CT$" and "$CP$" are represented by the current locations of the movement UTA and task-execution UTA, respectively. "$ST$" is declared as a two-dimensional array of Boolean variables that stores the execution statuses of tasks ($EST$), that is, *finished* (*true*) or *unfinished* (*false*), for all agents in the environment. "$TT$" can be the index of the target milestone, or the index of the next task.

Note that a *Q-state* does not contain continuous variables such as clocks, because it is impossible to sample all the possible values of continuous variables in the simulation. Moreover, the mission-planning problem concerns the *EST* of agents, which are covered by *Q-states* already. Introducing other variables, e.g., a global clock variable that measures the entire time of mission execution, would be redundant. In addition, to symbolise Q-states with clock variables, we need to use *zones* [20] instead of their concrete values, which complicates the problem unnecessarily. The existence of "*ST*" in *Q-states* requires the agents to communicate with each other, which introduces overhead and unreliability in the implementation of these agents. However, to solve the mission-planning problem with uncertain task execution times, this cost is necessary.

To apply Q-learning, we need to define a formula to calculate the rewards for state-action pairs. The rewards should encourage the agents to accomplish their tasks as fast as possible. Hence, a global clock variable named `gt` that measures the total execution time of agents is defined in our UTA model, although it is not included in the *Q-states*. UPPAAL STRATEGO provides a special query [25] that allows us to simulate the model, sample the specific runs, and pass them to the learning algorithm (e.g., Q-learning):

```
strategy opt = minE(x)[<=T]{dv}->{cv}:<> P
```
(9.5)

In Query (9.5), `minE(x)` simulates the model while executing the learning algorithm to minimize "x", which can be a variable or an expression. Parameter `T` is the maximum simulation time, `dv` is a set of discrete variables, and `cv` is a set of continuous variables. The learning algorithm observes the state space of the model partially, by detecting the values of the variables in `dv` and `cv`. The formula "`<>P`" is a (T)CTL property satisfied by the runs sampled from the simulation. These runs are used as input to the learning algorithm to evaluate state-action pairs. In this mission-planning problem, the global clock variable `gt` is `x`, the attributes of *Q-state* constitute `dv`, `cv` is an empty set, and `P` is as follows, being also used in Query (9.4):

$$\texttt{(forall(i:int[0,N-1]) ite[i]} \geq \texttt{M) \&\& gt} \leq \texttt{TL} \tag{9.6}$$

Algorithm 1 presents the process of executing queries in the form of Query (9.5) in UPPAAL STRATEGO. Parameters *stg, iterationNum, totalNum*, and *goodNum, formula* represent the STG model, the user-defined number of iterations of learning, the user-defined maximum rounds of simulation, the maximum number of runs that satisfy the property, and the property (`<> P` in Query (9.5)), respectively.

**Algorithm 1:** Simplified algorithm behind the `minE`-query of UPPAAL STRATEGO

```
1  Main(stg, iterationNum, totalNum, goodNum, formula)
2  int iterations = 0
3  int bestFitness = ∞
4  Strategy best = empty
5  Strategy aStrategy = empty
6  for iterations < iterationNum do
7      int totalRuns = 0
8      int goodRuns = 0
9      for totalRuns < totalNum do
10         Run aRun = simulate(stg, aStrategy)
11         if aRun satisfies formula then
12             aStrategy = learn(aRun)
13             goodRuns + +
14             if goodRuns ≥ goodNum then
15                 break
16         totalRuns + +;
17     if goodRuns ≥ goodNum then
18         fitness = evaluate(aStrategy)
19         if fitness < bestFitness then
20             bestFitness = fitness
21             best = aStrategy
22     iterations + +
23 return best;
```

At lines 4 and 5 of Algorithm 1, two empty strategies are defined, which are two arrays for storing Q-tables, in practice. In line 10, random simulation starts, from which the runs that satisfy "`<> P`" (a.k.a., good runs) are sent to the learning algorithm (line 12), which can be an internal function of UPPAAL STRATEGO or a pre-compiled library. The check of satisfaction of "`<> P`" is done by UPPAAL STRATEGO. For details, we refer the interested reader to the literature [14]. The learning algorithm calculates the rewards of the state-action pairs in these good runs based on the value of "`x`", and stores the rewards in the variable *aStrategy* (line 12).

The simulation and learning terminate under two conditions: (i) when the total rounds of simulation reach the limit (line 9), or (ii) when the number of good runs reaches the limit (line 15). When the simulation terminates in case (i), no strategy is generated as the good runs collected from the simulation do not support generating a complete strategy; if the simulation terminates in case (ii), a strategy is generated and stored as a *Q-table*. Lines 17 to 21 evaluate the learned strategy. The *fitness* of a strategy is the expectation of "`x`" when the model is under the control of the strategy up to the horizon provided in the query. If the query is `minE` (respectively, `maxE`), the evaluation observes the fitness of the current strategy and judges if its value is less (respectively, larger) than the value of the best strategy, and updates the best strategy accordingly.

### Verification of the Synthesized Strategies

As depicted in Fig. 9.9, after a strategy is synthesized, the model checker of UPPAAL STRATEGO is employed to verify the TG of the system under the control of the strategy. UPPAAL STRATEGO provides a special query to realize this function, which is shown in Query (9.7), where `P` is a Boolean expression, e.g., Query (9.6), `opt` is the strategy that is synthesized by Query (9.5) and that controls the behavior of the model:

```
A<> P under opt
```
(9.7)

Specifically, when UPPAAL STRATEGO reaches a state where it faces multiple controllable actions, the model checker can filter out non-optimal choices (according to the strategy) from the exploration of the system.

We extend UPPAAL STRATEGO such that a subset of strategies generated by Query (9.5) can also be verified by Query (9.7). This is an extension of the original work on UPPAAL STRATEGO [14] where only strategies generated by the game-theoretic synthesis of UPPAAL TIGA [8] can be verified. The subset of strategies here refers to the ones that do not have clock variables. As defined in Definitions 5 and 6, the strategies (i.e., Q-tables) of MCRL do not contain clocks.

This verification is step 4 of the method (see Fig. 9.10). For the users of this method, synthesizing a strategy and verifying it are two consecutive operations of running Queries (9.5) and (9.7). However, there are two important steps of model conversion that are executed underneath, by the tool: *probabilistic quantification* and *abstraction* (Fig. 9.10). Probabilistic quantification is explained in Section 9.6.2. Now we introduce the abstraction from stochastic strategies $\sigma^\circ$ of STG to strategies $\sigma$ of the corresponding TG. As stated in Definitions 4 and 2, $\sigma^\circ$ assigns probabilities to the controllable actions of the agents, whereas $\sigma$ explicitly informs the agents what is the next action to do at each state.

In practice, given a Q-table that contains the values of state-action pairs defined in Definitions 5 and 6, we construct strategies $\sigma$ by using the rewards as the priorities of choosing actions at the corresponding states, that is, the actions with the highest values are always chosen by the model checker. When multiple actions have the same value at some states, the model checker will exhaustively select each one of them to execute and check, in a non-deterministic manner. In this way, we can verify if the strategies synthesized by Q-learning are guaranteed to be complete and correct in the sense that the new models of the agents, which are controlled by the strategies, satisfy the requirements considering all the possible task execution times. In addition, the state-space explosion of the original TAMAA is overcome, since the state space that is explored by the model checker for verifying the new models is much reduced by the strategies.

To guarantee that the synthesized strategies meet all the requirements mentioned in Section 9.3.2, we design queries as presented below. In these queries, $\texttt{te}_n$ and $\texttt{move}_n$ are the task execution TG and movement TG of agent $n$, respectively. The variable `tf` is a two-dimensional Boolean array of agents' task execution statuses, e.g., finished, or unfinished, `x` is a clock variable, and `opt` is the synthesized strategy.

- *Milestone Matching.* Query (9.8) checks that agent's $n$ position is always at milestone $P_i$, when it is executing task $T_i$:

$$\texttt{A[] (te}_n\texttt{.T}_i\texttt{ imply move}_n\texttt{.P}_i\texttt{) under opt} \tag{9.8}$$

- *Task Sequencing.* Query (9.9) checks if the precedent task $T_{i-1}$ is always finished, when agent $n$ is executing task $T_i$:

$$\texttt{A[] (te}_n\texttt{.T}_i\texttt{ imply tf[n][i-1]==true) under opt} \tag{9.9}$$

- *Timing.* Query (9.10) checks if the agents can always finish all their tasks within *TL* time units, where `N` is the number of agents, `M` is the requested tasks iteration number, and `TL` is an integer of time limit:

```
A<> ((forall(i:int[0,N-1]) fin[i]≥ M)
                    imply x≤TL) under opt          (9.10)
```

## 9.7 Tool Support

In this section, we describe the automated support for our method, our toolset *MALTA*[5], which is depicted in Fig. 9.11. A GUI named *Mission*

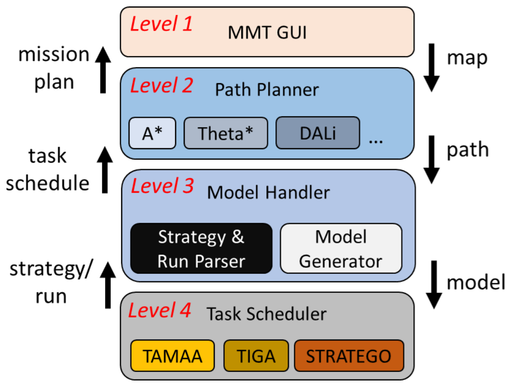


Figure 9.11: The structure of the toolset.

*Management Tool* (MMT) is designed at the top level to enable users to configure the map, agents, tasks, and milestones, etc., capturing the information of the environment. A module named *Path Planner* is designed at the second level, to support various path-planning algorithms, e.g., A* [2], Theta* [3], and DALi [26]. The *Path Planner* obtains the information of the map, including the navigation area, forbidden areas, milestones, etc., and calculates paths among the milestones and avoid all the forbidden areas. DALi can even select paths intelligently, when

[5]MALTA is published: https://github.com/rgu01/MALTA.

encountering temporary obstacles, crowed areas, etc. We refer the reader to literature [26] for details.

In the experiments of this paper, we use Theta* for path planning, due to its capability of calculating smooth paths that minimize the amount of sharp turns. The *Path Planner* sends the paths to the third level, where a module named *Model Generator* is designed to produce the TG/UTA models of agents, automatically. These models are used in the fourth level called *Task Scheduler*, which invokes TAMAA, UPPAAL TIGA, or UPPAAL STRATEGO, based on the requirement and scale of the problem of synthesizing strategies. Strategies (respectively, runs), synthesized by UPPAAL TIGA or UPPAAL STRATEGO (respectively, TAMAA), are then sent back to the third level, where a module named *Strategy & Run Parser* is designed to parse the strategy or runs into the format that is understandable for the second level. Last, the task schedule and the path plan are combined as a mission plan and shown in MMT GUI.

A detailed description of levels 1 to 3 of the toolset can be found in previous work [7]. We focus on level 4, *Task Scheduler*, in this section. In our previous work [12], we have proposed an implementation of MCRL, which uses the *simulation* query in UPPAAL SMC to randomly simulate the models, gathers enough runs that satisfy a condition, and prints the rewards of the state-action pairs of the runs into text files. Next, the files are parsed and used as the source data for the Q-learning algorithm to populate Q-tables. The Q-tables are then injected back into the models. A new UTA named *conductor* is designed to read the Q-tables every time when the agent needs to make a decision. The *conductor* sends signals to the movement and task-execution UTA, in order to control them to perform different actions according to the Q-table.

This implementation separates the data-gathering phase from the learning phase, so the rewards of state-action pairs accumulated in the data-gathering phase cannot easily be exploited for guiding the sampling in a strategic manner. In every round of simulation, the *simulator* explores the state space randomly, with unchanged probabilities of the actions. Moreover, the UTA models allow the most permissive behaviors of agents, such as wondering without executing any tasks. The separation of phases makes the simulation unlikely to reach the states where rare events happen, e.g., multiple iterations of tasks, or finishing a large number of tasks in a strict time frame. Therefore, the new version of the method embeds MCRL into UPPAAL STRATEGO to fix this inconvenience, which will be introduced in the next subsection.

### 9.7.1 Integration of Task Scheduler and UPPAAL STRATEGO

As shown in Algorithm 1, in the new version of MCRL, once a run that satisfies our requirement is obtained from the simulation (line 11), it is directly fed into the learning algorithm to synthesize a strategy (line 12). The strategy is not necessarily complete, but it is then input into the next round of simulation (line 10), where the *simulator* can exploit the existing strategy by using the rewards accumulated in the past rounds of simulation as the probabilities of actions (see Subsection 9.6.2). Therefore, the actions with higher rewards will be chosen more likely than the ones with lower rewards, and thus, the learning phase is accelerated.

After a certain rounds of simulation (the number is configurable), a candidate strategy is produced and sent to the model checker to verify if it guarantees to enable the agents to finish all tasks according to the requirement, regardless of how the environment reacts. This process iterates until the verification passes. In our previous implementation of MCRL [12], the movement and task-execution UTA are modified, and a UTA named *conductor* is created to control the models according to the Q-table. In our current implementation, the original movement and task-execution TG are directly verified in UPPAAL STRATEGO by running queries in the format of Query (9.7). When UPPAAL STRATEGO verifies the models against these queries, it calls back a function in the external library of the learning algorithm, where the Q-table is stored, whenever it faces multiple available controllable actions. This function searches the Q-table and returns the highest priority for the actions with the highest rewards to the model checker. UPPAAL STRATEGO then exhaustively explores the actions that have the same highest priority, but ignores the ones with lower priorities. In this way, the models' behavior is under the control of the strategy without introducing new models, such as a *conductor* UTA.

Additionally, the new version of MCRL is implemented in an extensible scheme. The learning algorithm is programmed in standard $C$ or $C++$, and compiled into an external library, which UPPAAL STRATEGO calls back when learning is required. Hence, the users of the tool can replace the Q-learning with their own learning algorithms, and leverage the formal aspects of the method to guarantee the completeness and correctness of the synthesized strategies.

## 9.8 Experimental Evaluation

In this section, we evaluate the improved version of TAMAA and the new version of MCRL in several experiments. The experiments are conducted on a laptop running an Intel Core i7 processor with 12 cores, 16 GB of RAM and a 64-bit Linux OS.

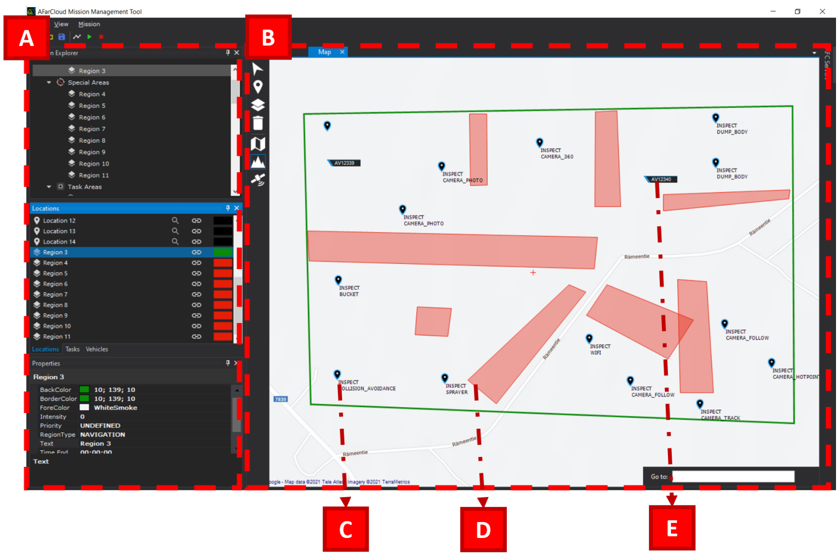
### 9.8.1 Design of Experiments

Fig. 9.12 depicts a working environment of agents created in MMT. Our mission planner calculates paths that enable the agents to visit the milestones in a certain order so that they finish their tasks in a correct and efficient way. According to previous investigation [7], the number of agents is the factor that impacts the computation time of the mission planners most profoundly. As shown in Table 9.2, UPPAAL TIGA could not handle more than 5 agents. Therefore, we vary the number of agents from 3 to 6 in the experiments in order to show that the new MCRL is capable of dealing with more agents than the improved version of TAMAA in UPPAAL TIGA. To demonstrate the extensibility of MCRL, the experiments are conducted on two versions of MCRL. One uses an external library of Q-learning and one uses the Q-learning function in UPPAAL STRATEGO [25], which are called external and internal Q-learning for brevity[6], respectively. We experiment with both an internal and an external version of Q-learning to study the impact of (1) a fully extensible implementation of the learning algorithm, and (2) the overhead of communication between UPPAAL STRATEGO and the external library. In addition, this construction allows us to define a custom strategy output format for an integration into our toolset *MALTA*. We also vary the number of milestones (correspondingly, tasks) to see how this factor influences the computation time.

### 9.8.2 Results of Experiments

Table 9.3 shows the numbers of explored states and computation time for the three mission planners synthesizing mission plans for 3, 4, 5, and 6

[6]Although the internal Q-learning is a part of UPPAAL STRATEGO, MCRL provides a post-verification to it and thus it is called MCRL with internal Q-learning.



| | |
|---|---|
| BackColor | 🟩 10; 139; 10 |
| BorderColor | 🟩 10; 139; 10 |
| ForeColor | ⬜ WhiteSmoke |
| Intensity | 0 |
| Priority | UNDEFINED |
| RegionType | NAVIGATION |
| Text | Region 3 |
| Time End | 00:00:00 |

Figure 9.12: A working environment of agents in MMT. Module $A$ is the configuration panel, where users configure the parameters of the map, vehicles, tasks, etc. Module $B$ is the map, where the environment is visualized. Synthesized mission plans will also be shown in this module. Pinpoints like $C$ are the milestones, where tasks are assigned to. Red areas like $D$ are the special areas, which can be fixed/temporary forbidden areas (a.k.a., static obstacles), crowed areas, etc. Tags like $E$ are the initial positions of agents.

agents, respectively. The number of milestones and tasks are fixed, such

Table 9.3: Explored states and computation time of synthesizing mission plans for different numbers of agents. The environment contains 3 milestones and 3 tasks.

| | States | Time | Agents |
|---|---|---|---|
| TIGA | 222,88 | 220 ms | 3 |
| MCRL with Internal Q-learning | 143,044 | 572 ms | |
| MCRL with External Q-learning | 428,550 | 2.0 s | |
| TIGA | 764,001 | 18.1 s | 4 |
| MCRL with Internal Q-learning | 772,619 | 2.2 s | |
| MCRL with External Q-learning | 1,150,349 | 7.3 s | |
| TIGA | 33,312,229 | 53.8 mins | 5 |
| MCRL with Internal Q-learning | 9,822,914 | 38.2 s | |
| MCRL with External Q-learning | 6,700,782 | 53.0 s | |
| TIGA | Out of memory | Unknown | 6 |
| MCRL with Internal Q-learning | 10,322,666 | 7.9 mins | |
| MCRL with External Q-learning | 100,901,760 | 14.8 mins | |

that the difference of the results among the mission planners would only be caused by the increased number of agents. We run the experiments 5 times for each mission planner in each scenario containing different number of agents, and use the mean values as the results. Clearly, UPPAAL TIGA can only deal with situations with less than 6 agents, whereas MCRL can cope with 6 agents within reasonable computation times: 7.9 minutes or 14.8 minutes. The difference in computation times of the two versions of MCRL is due to the different implementations of Q-learning and the overhead of communication between UPPAAL STRATEGO and the external library. They collectively confirm the conclusion that MCRL outperforms TAMAA when the number of agents is large.

As depicted in Table 9.4, strategies synthesized by MCRL with the external and internal Q-learning are complete in the sense that they satisfy liveness queries in the form of Query (9.7). When the number of agents is greater than 4, internal Q-learning needs more simulation rounds to sample enough runs for learning than that of the external

Q-learning. The reason for this is discussed in the next subsection.

Table 9.4: The numbers of sampled traces and total simulation rounds that are needed for synthesizing strategies by using the MCRL with the internal and external Q-learning, as well as the completeness of the synthesized strategies.

| | Sampled traces | Total runs | Completeness | Agents |
|---|---|---|---|---|
| External Q-learning | 100 | 2,000 | true | 4 |
| Internal Q-learning | 100 | 2,000 | true | |
| External Q-learning | 200 | 10,000 | true | 5 |
| Internal Q-learning | 200 | 20,000 | true | |
| External Q-learning | 200 | 100,000 | true | 6 |
| Internal Q-learning | 200 | 150,000 | true | |

Table 9.5 shows the number of explored states and computation time for the three mission planners synthesizing mission plans for 2 agents, but different numbers of milestones and tasks (tasks are assigned to milestones, thus N milestones imply N tasks). As presented in the table, the number of explored states and computation time of UPPAAL TIGA do not increase very fast with the increasing numbers of milestones and tasks, which is consistent with our previous investigation [7]. However, two versions of MCRL with the internal and external Q-learning perform much worse than UPPAAL TIGA when the numbers of milestones and tasks are greater than 5. Note that, when the numbers of milestones and tasks are 10, the rates of synthesizing complete strategies by using the external and internal Q-learning are lower than 10%. We discuss the reason for this result in the next subsection.

### 9.8.3 Discussion of the Experimental Results

As MCRL randomly searches the state space multiple times during the learning process, the numbers of explored states of these two planners do not reflect the size of the agent model. Hence, we compare the numbers of explored states obtained with UPPAAL TIGA in Table 9.3 and Table

Table 9.5: Explored states and computation time of three methods synthesizing mission plans for different numbers of milestones and tasks. The environment contains 2 agents.

|  | States | Time | milestones & tasks |
|---|---|---|---|
| TIGA | 11,746 | 61 ms | 5 |
| MCRL Internal Q-learning | 136,113 | 347 ms | |
| MCRL External Q-learning | 200,963 | 1.1 s | |
| TIGA | 161,953 | 1 s | 8 |
| MCRL Internal Q-learning | 49,489,463 | 3.4 mins | |
| MCRL External Q-learning | 63,858,459 | 8.2 mins | |
| TIGA | 586,124 | 3.9 s | 10 |
| MCRL Internal Q-learning | 324,257,087 | 33.7 mins | |
| MCRL External Q-learning | 324,283,558 | 46.4 mins | |

9.5, and conclude that the size of the state space of the agent model is mainly influenced by the number of agents. The numbers of milestones and tasks increase the state space much less significantly, but the trend of increase is still exponential.

The reason why MCRL with the external Q-learning needs less total rounds of simulation than that of the internal Q-learning is because the intermediary strategies are adopted in the external Q-learning during the course of simulation for a heuristic exploration of the state space. The difference of heuristic exploration in two versions of MCRL results in the different requested rounds of simulation, which also contributes to the worse performance of the simulation-based algorithms when the numbers of milestones and tasks are more than 8.

Currently, the learning algorithms can only leverage the “good” runs that satisfy our requested condition (e.g., Formula (9.6)). When the milestones and tasks are few, random simulation can easily get to the states where the property holds. As the numbers of milestones and tasks increase, it becomes increasingly unlikely to reach the terminal state by random simulations, which in turns implies that a guided search cannot occur in the simulation. Runs that do not satisfy the specified condition are not provided to the learning algorithm, and thus, do not contribute to the heuristic exploration of the state space. Therefore, in cases with large numbers of milestones and tasks, large numbers of simulation rounds are needed to generate enough "good" runs, which result in a high number of explored states and a long computation time. We leave the improvement of the method for future work, but hypothesize that the inclusion of negative reinforcement feedback (that is, runs that do not meet the goal will receive a penalty) will improve the performance of MCRL significantly.

In summary, when the number of agents is large, MCRL with the internal Q-learning is the first option because it scales better than UPPAAL TIGA and needs less computation time than MCRL with the external Q-learning does in this case. In the cases where the numbers of milestones and tasks are large, UPPAAL TIGA is the first option as it scales and provides strategies that are guaranteed to cover all possible scenarios. When the users need to embed their own learning algorithms in the method, MCRL with the external Q-learning is the first option because the learning module is an external library that can be replaced easily.

## 9.9 Related Work

Synthesis of strategies for multiple autonomous agents has become an increasingly studied area. Wang et al. [27] attempt to address the scalability challenge of solving POMDP (Partially Observable Markov Decision Processes) with safe-reachability objectives. Similar to the bounded model-checking technique [28], their method constrains the state-space of the model by using a goal-constrained belief space instead of the entire belief space. Bouton et al. [29] focus on a concrete scenario of autonomous cars: navigation in unsignalized intersections. Their method is based on POMDP and Monte Carlo sampling, thus avoiding the scalability problem. However, their method does not provide formal-verification-based guarantees of correctness. Nikou et al. [30] propose a solution to synthesize controllers of agents for path planning. Their synthesized controllers also satisfy complex high-level constraints of tasks. However, no proof of scalability with the number of agents is provided. Our approach is accompanied by a toolset that is capable of handling mission-plan synthesis for multiple agents, mitigating the associated lack of scalability caused by the numbers of agents, milestones, and tasks.

Similar to our work, some studies also combine formal verification with learning algorithms. The UPPAAL STRATEGO [14] tool facilitates both sample-based optimization and correct-by-construction controller synthesis. In addition, both these methods can be combined for safe and (near-)optimal synthesis. Basile et al. [31] use UPPAAL STRATEGO to solve the strategy synthesis problem for autonomous driving in a moving block railway system. They leverage the game-theoretic method to synthesize safe strategies and reinforcement learning to optimize the strategies. To achieve formal correctness of a learned controller, UPPAAL STRATEGO relies on learning under a prior construction of the safe controller, specifically guarding the learning against unsound actions. This is contrary to our simulation-based approach (namely, MCRL) where learning is conducted on the original models directly to synthesize strategies with no guarantee of correctness. The post-verification in MCRL adds correctness guarantees on the learned strategies, which eliminates the state-space explosion problem that exists in the original models.

Similar to TAMAA, the approach of Gleirscher et al. [32] is also based on graphic search. Their approach is able to synthesize and verify safety controllers for human-robot collaboration. Bersani et al. [33] present the PuRSUE (Planner for RobotS in Uncontrollable Environments) approach, which supports users to configure their robotic applications and automatically generate their controllers by using UPPAAL TIGA. The main difference between MCRL and theirs is that their synthesis is based on graphic search and thus limited on scalability.

Li et al. [34] focus on capturing complex and domain-specific requirements of robotic systems by using formal specification languages. Their method also makes the reward generation of the learning process interpretable and guarantees the satisfaction of specification, for critical components of the systems. The method proposed by Bouton et al. [35] enforces probabilistic guarantees on agents during the course of reinforcement learning. Brázdil et al. [36] provide algorithms for searching MDP (Markov Decision Processes) to verify various reachability properties. Legay et al. [37] present a scalable approach of verification for MDP. When comparing to these studies, we apply model checking on the learned strategies and facilitate the verification for complex models with large state spaces by using reinforcement learning, rather than constructing initially a safe restriction of the system. Our work is orthogonal to that of Brázdil, Legay and Bouton, that is, their methods could be utilized for the initial construction of strategies to demonstrate the non-existence of rare events. Our method can be then used to verify their strategies. In addition, our method has the ability to handle timed systems and distributions over durations.

To the best of our knowledge, the earliest attempt to employ reinforcement learning for solving the state-space-explosion problem of model checking is done by Behjati et al. [38]. The authors propose a bounded rational verification approach for on-the-fly model checking. However, this method is limited to LTL properties, and it has not been applied on autonomous agents.

## 9.10 Conclusions and Future Work

In this paper, we have presented our method of solving the mission-plan synthesis problem of multiple autonomous agents. The method is based on our tool named TAMAA and improves the original TAMAA with the ability of handling uncertain movement time and task execution time of agents. Additionally, our method, called MCRL, combines model checking with reinforcement learning, so that it is capable of dealing with more agents than the improved TAMAA, which applies model checking alone. MCRL provides a means for verifying and analyzing the synthesized mission plans by using model checking, to ensure that safety-critical requirements are met. The method is fully integrated with UPPAAL STRATEGO. We demonstrate MCRL's ability of handling multiple agents by experiments, and compare the result with the original and improved TAMAA. The number of explored states and computation time of MCRL increase much slower than the two versions of TAMAA when the number of agents increases. However, the improved version of TAMAA in UPPAAL TIGA performs better than MCRL when the number of agents is less than two but the numbers of milestones and tasks are more than five.

One of the future directions of work is to improve the learning algorithm of MCRL to perform better in environments with large numbers of milestones and tasks. Another future work direction focuses on estimating the existence of strategies of timed games before synthesis starts. Introducing variables that evolve continuously, e.g., time, energy consumption, in the models and strategies is another interesting direction of future research, which would dramatically complicate the strategy-synthesis problem.

## Acknowledgments

We acknowledge the support of the Swedish Knowledge Foundation via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr: 20150022, and via the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190-038.

# 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*. Springer, 1996.

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

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

[4] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report*, 1998.

[5] Henry Fisher. Probabilistic learning combinations of local job-shop scheduling rules. In *Industrial scheduling*. Englewood Cliffs, NJ: Prentice Hall., 1963.

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

[7] Rong Gu, Eduard Paul Enoiu, and Cristina Seceleanu. TAMAA: UPPAAL-based mission planning for autonomous agents. In *ACM/SIGAPP Symposium On Applied Computing*, 2020.

[8] Gerd Behrmann, Alexandre David, Emmanuel Fleury, Kim Larsen, Didier Lime, and Ecole Nantes. Uppaal-Tiga: Time for playing games! (tool paper). In *International Conference on Computer Aided Verification*. Springer Berlin Heidelberg, 2007.

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

[10] Radek Pelánek. Fighting state space explosion: Review and evaluation. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2008.

[11] Frederik Meyer Bønneland, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marco Muñiz, and Jiří Srba. Stubborn set reduction for two-player reachability games. *arXiv preprint arXiv:1912.09875*, 2019.

[12] Rong Gu, Eduard Paul Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Verifiable and scalable mission-plan synthesis for multiple autonomous agents. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2020.

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

[14] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal Stratego. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015.

[15] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. King's College, Cambridge United Kingdom, 1989.

[16] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126, 1994.

[17] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *International Conference on Concurrency Theory*. Springer, 2005.

[18] Alexandre David, Peter G Jensen, Kim Guldstrand Larsen, Axel Legay, Didier Lime, Mathias Grund Sørensen, and Jakob H Taankvist. On time with minimal expected cost! In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2014.

[19] Jean-Francois Kempf, Marius Bozga, and Oded Maler. As soon as probable: Optimal scheduling under stochastic uncertainty. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013.

[20] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, 2004.

[21] 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.

[22] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, 2000.

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

[24] Rong Gu, Cristina Seceleanu, Eduard Paul Enoiu, and Kristina Lundqvist. Model checking collision avoidance of nonlinear autonomous vehicle models. In *Formal Methods 2021*, 2021.

[25] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2019.

[26] Luigi Palopoli, Antonis Argyros, Josef Birchbauer, Alessio Colombo, Daniele Fontanelli, Axel Legay, Andrea Garulli, Antonello Giannitrapani, David Macii, Federico Moro, et al. Navigation assistance and guidance of older adults across complex public spaces: the DALi approach. *Intelligent Service Robotics*, 2015.

[27] 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*. Springer, 2018.

[28] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. *Bounded model checking.* Carnegie Mellon University, 2003.

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

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

[31] Davide Basile, Maurice H ter Beek, and Axel Legay. Strategy synthesis for autonomous driving in a moving block railway system with uppaal stratego. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems.* Springer, 2020.

[32] Mario Gleirscher, Radu Calinescu, James Douthwaite, Benjamin Lesage, Colin Paterson, Jonathan Aitken, Rob Alexander, and James Law. Verified synthesis of optimal safety controllers for human-robot collaboration. *arXiv preprint arXiv:2106.06604*, 2021.

[33] Marcello M Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione, and Matteo Rossi. Pursue-from specification of robotic environments to synthesis of controllers. *Formal Aspects of Computing*, 2020.

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

[35] 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.

[36] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtěch Forejt, Jan Křetínský, Marta Kwiatkowska, David Parker, and Mateusz Ujma. Verification of markov decision processes using learning algorithms. In *International Symposium on Automated Technology for Verification and Analysis.* Springer, 2014.

[37] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Scalable verification of markov decision processes. In *International Conference on Software Engineering and Formal Methods*. Springer, 2014.

[38] Razieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. Bounded rational search for on-the-fly model checking of LTL properties. In *Symposium on the Foundations of Software Engineering*. Springer, 2009.

# Chapter 10

# Paper C: Synthesis and Verification of Mission Plans for Multiple Autonomous Agents under Complex Road Conditions

Rong Gu, Eduard Baranov, Afshin Ameri, Eduard Enoiu, Baran Cürüklü, Cristina Seceleanu, Axel Legay, and Kristina Lundqvist
Submitted to Transactions on Software Engineering and Methodology, ACM, 2022.

## Abstract

Mission planning for multi-agent autonomous systems aims to generate feasible and optimal mission plans that satisfy the given requirements. In this article, we propose a mission-planning methodology that combines (i) a path-planning algorithm for synthesizing path plans that are safe in environments with complex road conditions, and (ii) a task-scheduling method for synthesizing task plans that schedule the tasks in the right and fastest order, taking into account the planned paths. The task-scheduling method is based on model checking, which provides means of automatically generating task execution orders that satisfy the requirements and ensure the correctness and efficiency of the plans by construction. We implement our approach in a tool named MALTA, which offers a user-friendly GUI for configuring mission requirements, a module for path planning, an integration with the model checker UPPAAL, and functions for automatic generation of formal models, and parsing of the execution traces of models. Experiments with the tool demonstrate its applicability and performance in various configurations of an industrial case study of an autonomous quarry. We also show the adaptability of our tool by employing it on a special case of the industrial case study.

## 10.1 Introduction

Autonomous robotic systems are becoming common in our society. These systems can take different forms, e.g. a vehicle that is used for transportation in a factory or in a construction site, a mobile robot used for entertainment in our homes, or a mobile communication platform used in the homes of the elderly. Obviously, these robotic systems are associated with different requirements, thus having different shapes and overall design. Despite the differences, these systems share a common feature: the ability to function in an environment without or with minimum human intervention. This feature can be referred to as *autonomous operation*. However, the environment where these systems operate in could include humans and other obstacles, hence it is unpredictable and only partially known to the system. To realize the autonomy feature, the autonomous robotic systems must be able to perceive the environment, reason based on known facts, and act to meet the requirements associated to their goals (for the sake of brevity autonomous robotic agents are called "autonomous agents", or simply "agents" in the rest of the paper [1]).

One key challenge of designing agents that move and operate in a confined environment is *mission planning* (a.k.a., mission plan synthesis), which includes *path planning* and *task scheduling*. When obstacles are present in the environment, the ability to reach a destination without colliding with them is a problem that has been solved by existing path-planning algorithms, such as A* [2] and rapidly-exploring random tree (RRT) [3] algorithms. However, the real environment might impose complex restrictions to path planning, stemming from obstacles that are temporary, or from existing special areas, like crowded or desired areas. Consequently, the *path-planning* algorithm should provide means to calculate a path plan for an agent that "wisely" chooses to wait, circumvent, or cross the respective areas. Furthermore, agents visit different positions (a.k.a. *milestones*) in the environment as part of their tasks, e.g., within a quarry, autonomous wheel loaders visit stones piles to load stones. Being able to guarantee that agents carry out the right tasks at the right milestones is important for the overall success of a mission. Additionally, tasks can have complex and temporal requirements, for instance, autonomous wheel-loaders must keep digging stones until trucks arrive, and then load the stones into the trucks before the latter transport the stones to crushers. Some applications require the agents to keep a certain level of productivity, e.g., autonomous trucks must transport all the stones to a crusher within a maximum time window of a few hours. Path-planning algorithms alone are not able to calculate *mission plans* that accomplish the tasks respecting such requirements. They must be combined with *task-scheduling* algorithms for synthesizing mission plans that ensure that the agents travel safely, that is, without any collision with static obstacles, and satisfy the requirements of tasks.

*Task-scheduling* algorithms aim to calculate an order of task execution to achieve the global goal, e.g., a pile of stones is dug and loaded, then transported to crushers, and then crushed into fractions. Based on the position of each agent, task scheduling assigns milestones and the visiting orders to the agents, respectively, so that the agents can finish their mission within a given time window. A classic presentation of this problem is called the *job-shop* problem, which is described as follows:

"*Given are a set of jobs and a set of machines, assume that: (i) Each machine can handle at most one job at a time, and (ii) Each job consists of a chain of operations, each of which needs to be processed during an uninterrupted time period of a given length on a given machine. The purpose is to find a schedule, i.e., an allocation of the operations to time intervals on the machines, that has minimum length.*" [4]

As an NP-hard problem, even a simple instance of the *job-shop* problem with very restrictive constraints remains difficult to solve [5]. Furthermore, task schedules that we aim to calculate must not only "have the minimum length", i.e., accomplish the tasks in the quickest way, but also satisfy the complex temporal requirements aforementioned. In a nutshell, we provide answers to the following research questions in this paper:

1. *Path planning*: considering an environment with various road conditions, such as static obstacles (e.g., forbidden areas and temporary obstacles), crowded areas, and desired/undesired areas, how to calculate a path that reaches the desired milestone without a collision?

2. *Task scheduling*: given a set of tasks and milestones where the tasks are supposed to be carried out, as well as a set of task execution constraints, how to synthesize a task-execution schedule that finishes tasks at their respective milestones and satisfies the task execution constraints?

3. *Mission planning*: how to combine path planning with task schedul-

ing and produce an optimal mission plan?

4. *Automation*: given a mission-planning problem that matches the problem definition (Definition 7), how to easily configure the scenario of the problem and automatically calculate mission plans that combine the results of path planning and task scheduling?

5. *Adaptability*: when the mission-planning problem does not match the problem definition, how to leverage the automation and easily adapt the models to solve the problem?

6. *Visualization*: how to visualize mission plans for demonstration purpose?

In this article, our main contribution is a methodology and a tool support for answering the research questions (1) - (6) collectively, with an optimal result, that is, not only a correct mission plan that meets various requirements but also reaches the goal in the fastest manner.

In particular, we adapt and tune up the DALI algorithm [6] for path planning, which takes into account both environmental constraints and user preferences (research question (1)). We adapt a method called *TAMAA (Timed-Automata based Mission planner for Autonomous Agents)* [7] for task scheduling, which uses the well-known timed automata (TA) [8] formalism for modeling, and model checking in UPPAAL [9] for generating results with a correctness guarantee (research question (2)). Specifically, TAMAA automatically generates a TA model of agents, including the movement TA and task execution TA, and checks the model to find the execution traces that reaches the goal state the fastest and satisfy other requirements. The contribution of the new version of TAMAA is the combination with DALI. Previously, TAMAA uses the result of path planning once and generates a mission plan that only considers the permanently existing obstacles. In this article, complex road conditions such as temporary obstacles are considered. Hence, the initially correct mission plans may become invalid when temporary obstacles are activated. Hence, we design a validator in MALTA to check if the temporary mission plan meets the complex road conditions, and an iteration of execution between DALI and TAMAA until a correct mission plan is produced (research question (3)).

The design and implementation of our methodology, i.e., a toolset named *MALTA*, answers to research questions (4) - (6). MALTA has a client-server architecture, which integrates a graphical user interface (GUI) called *MMT* [10] for mission management in the client, data exchange modules, and path-planning and task-scheduling algorithms in the server. The client side of MALTA is MMT where users can configure the environment, missions for the agents, and parameters of agents, like their speeds, respectively. The server side has two modules: middleware and a back end. The middleware is responsible for obtaining the mission information from the GUI, running a path-planning algorithm, and generating agent models by using the path-planning results and mission information. Next, the middleware sends the agent models to the back end, where TAMAA runs for task scheduling. As our methodology is designed to cope with complex road conditions, such as temporary obstacles, MALTA possibly runs more than one iteration rounds between the middleware and the back end until an optimal mission plan is produced.

If a mission plan exists, it is visualized by MALTA so that users can check the details of the mission plan, such as when and which agent starts to execute a certain task, and how temporary obstacles can affect the plan (research question (6)). MALTA generates models and parses traces automatically, which eases the work of model and mission plan construction, especially for the cases where the amount of agents or the size of the environment is large (research question (4)). One can also modify the models according to one's own applications and still enjoy the facility of other automation provided by MALTA, such as trace parsing and information extraction from the map (research question (5)). MALTA has been applied on an industrial use case of an autonomous quarry that exposes the challenges of synthesizing time-optimal mission plans of several autonomous vehicles with various requirements.

The rest of the paper is organized as follows. In Section 10.2, we introduce the industrial case study. Section 10.3 describes the preliminaries including timed automata and UPPAAL, and the DALI algorithm for path planning. Our methodology for solving the mission-planning problem for multiple agents is introduced in Section 10.4, followed by the description of the toolset in Section 10.5. In Section 10.6, we conduct experiments on a normal use case taken from the industrial case study, which matches our problem definition, for the evaluation of our methodology. Section 10.7 demonstrates how the method can be adapted to a special use case of our industrial case study. In Section 10.8, we present and compare to related work, whereas in Section 10.9, we conclude the

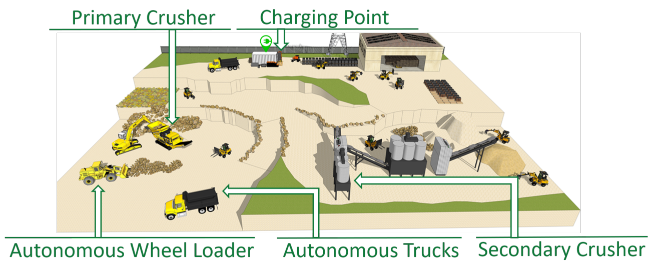


Figure 10.1: An example of an autonomous quarry

paper and outline some directions of future work.

## 10.2 An Industrial Case Study: The Autonomous Quarry

In this section, we introduce an industrial use case of an autonomous quarry provided by VOLVO Construction Equipment (CE) in Sweden. This use case serves as a running example through Sections 10.2 to 10.6, and as concrete motivation for our research. In the quarry, there are several stationary machines and vehicles. Examples of stationary machines are crushers that crush stones into certain sizes and are controlled manually. Typical examples of vehicles would be wheel loaders that dig stones, and autonomous trucks that transport stones into crushers. We assume that the wheel loaders are controlled manually and focus on the synthesizing mission plans for the autonomous trucks that are not necessarily identical, which means they can have different speed limits (e.g., 50 − 80 km/hour) and capability of transportation (e.g., 10 − 50 $m^3$). The vision is to deploy the autonomous trucks that performs certain tasks in order to fulfill the objectives defined by the operators of the quarry. The collection of such tasks together define a mission.

A simple quarry is illustrated in Figure 10.1. In this example, the stones should be dug up and loaded into the autonomous trucks by wheel loaders. Then trucks transport the stones and unload them into a primary crusher first, then later to a secondary crusher, the destination of the stones. Trucks must go to a charging pole when the battery level is low. The milestones are the positions where the tasks are carried out, e.g., at stone piles, crushers, and charging poles. A mission in this use case can be to dig, crush, and transport $1000m^3$ of stones in 24 hours.

Vehicle transportation should consider the environment. There could be static obstacles, that is, areas that are impassable by the vehicles (buildings, sizable holes in the road, etc.), and different road conditions that may cause the vehicles to slow down, e.g. muddy or bumpy roads, or form a temporarily inaccessible area, such as, during human intervention for maintenance, the corresponding area should not be crossed by any vehicle. Navigation must ensure a collision-free transportation in the presence of all the environmental constraints. The execution of tasks, e.g., loading, unloading, and charging, must be scheduled correctly and efficiently such that the trucks are guaranteed to accomplish the mission. In the autonomous quarry, all the planning work must be done automatically. Therefore, we need a planning methodology and a tool of planning that ensures completion of the mission and meeting additional requirements.

Based on this use case we formulate several requirements. We group the requirements for the *task scheduling* into the following categories:

- **Requirement Category I** (*milestone matching*): Tasks must be executed at the correct milestones, e.g., loading stones into primary crushers must be executed at a primary crusher.
- **Requirement Category II** (*task sequence*): Tasks must be executed in the right order, e.g., loading stones from stone piles must precede unloading stones into crushers.
- **Requirement Category III** (*timing*): Tasks can be executed multiple times and must be finished within a time frame in order to maintain a certain level of productivity, e.g., quarrying $1500m^3$ stones per day.

*Path planning* takes care of finding safe and fast paths for agents between milestones in complex environments. We consider the following types of environmental abnormalities that should be taken into account by the path planner:

- **Environmental Abnormality I** (*obstacles*): permanent and temporary obstacles must be avoided by agents. Permanent obstacles are always present, while temporary ones appear at known time points and disappear later on. Within this work we assume that all obstacles are static.

- **Environmental Abnormality II** (*road conditions*): muddy or bumpy roads cannot be passed at the full speed. The planner must decide whether it is faster to take a detour than to travel on these roads slowly.
- **Environmental Abnormality III** (*soft constraints*): for some reasons, e.g., safety, some areas might be considered undesirable for driving, and the planner must attempt to avoid such areas if there exists an alternative path. Such constraints should be satisfied unless they prevent the satisfaction of other requirements with higher priority. In the latter case, soft constraints can be ignored.

Note that path planning and task scheduling are not independent. Traveling time among milestones are needed by a task scheduler, while the presence of temporary obstacles can influence the starting time of travels. Therefore, path planning and task scheduling must work together to produce efficient mission plans.

In this paper, we work only with static scheduling: generated mission plans cannot change without full recomputation, therefore any additional tasks or unpredictable events are not considered, including unpredictable moving obstacles. Path following and the collision avoidance between agents are also outside the scope of this paper. We refer the interested reader to the literature [11] for these topics. In addition, the order of visiting milestones only depends on the constraints of task sequence. For example, for a mission containing two tasks $A$ and $B$ that must be carried out at milestones $a$ and $b$, respectively, the order of visiting the milestones only depends on the task sequence of $A$ and $B$. In another word, if tasks $A$ and $B$ can be executed in any order, then visiting milestone $a$ and $b$ can be any order too. To define the scope of automation of our methodology, we informally define the mission-planning problem and its solution as follows. The set of (non-negative) real numbers is denoted as $\mathbb{R}$ ($\mathbb{R}_{\geq 0}$).

**Definition 7** (Mission Planning)**.** *A mission-planning problem is a tuple:*

$$\mathcal{P} = <\mathcal{E},\ Ab,\ \mathcal{M},\ \mathcal{T},\ f,\ Req>, \tag{10.1}$$

*where $\mathcal{E} \subset \mathbb{R}^n$ is a confined environment, $n \in \{2, 3\}$, $Ab \subseteq \mathcal{E}$ is a set of areas that belong to one of the Environmental Abnormalities I, II, and III, $\mathcal{M} \subset \mathcal{E}$ is a set of milestones, $\mathcal{T}$ is a set of tasks, $f : \mathcal{M} \to \mathcal{T}$ assigns* *each of the tasks to one or multiple milestones, and Req is a set of task requirements that matches the Requirement Categories I, II, and III.*

Assuming an agent is equipped with the ability of doing two types of actions: moving to a milestone and executing a task, a solution of the mission-planning problem is informally defined as follows:

**Definition 8** (Solution of Mission Planning)**.** *Given a set of autonomous agents $\mathcal{V}$, a solution that enables the agents in $\mathcal{V}$ to solve a mission-planning problem $\mathcal{P} =< \mathcal{E},\ Ab,\ \mathcal{M},\ \mathcal{T},\ f,\ Req >$ is a tuple:*

$$plan =< schedule,\ path >, \tag{10.2}$$

*where schedule is a set of pairs $(st,\ ft)$, $st,\ ft \in \mathbb{R}_{\geq 0}$ are the time points of starting and finishing the agents' actions, respectively, path is a set of sequences of points $p \in \mathcal{E}$ that the agents must follow in order to reach the milestones $m \in \mathcal{M}$. Let $E1$, $E2$, and $E3$ be the sets of environment constraints belonging to Environmental Abnormalities I, II, and III, respectively, s.t., $Ab = E1 \cup E2 \cup E3$, and $E3' \subseteq E3$ be the subset of $E3$, which does not contradict $E1$ and $E2$, that is, the paths that meet $E3'$ do not violate $E1$ and $E2$, plan must hold the following two conditions:*

- *$\forall e \in E1 \cup E2 \cup E3'$, $path \models a$, that is, the paths must meet environmental constraints in $E1$, $E2$, and $E3'$,*
- *$\forall r \in Req$, $schedule \models r$, that is, the task schedule must satisfy all the requirements in Req.*

In the rest of this paper, we introduce our methodology for automatically generating solutions of mission-planning problems, defined in Definitions 8 and 7, respectively. We also demonstrate the adaptability of the methodology to solve problems that does not match Definition 7 with a slight change of the models.

## 10.3 Preliminaries

In this section, we introduce UPPAAL - the modeling, simulation, and verification tool that uses *Timed Automata* as the modeling formalism, which we apply for modeling agents movement and task execution. We also briefly describe the path-planning algorithm DALI that we employ in MALTA. We denote the set of natural numbers as $\mathbb{N}$.

### 10.3.1 Timed Automata and UPPAAL

**Definition 9.** *A* Timed Automaton *(TA) [8] is a tuple:*

$$\mathcal{A} =< L, l_0, X, \Sigma, E, Inv >, \tag{10.3}$$

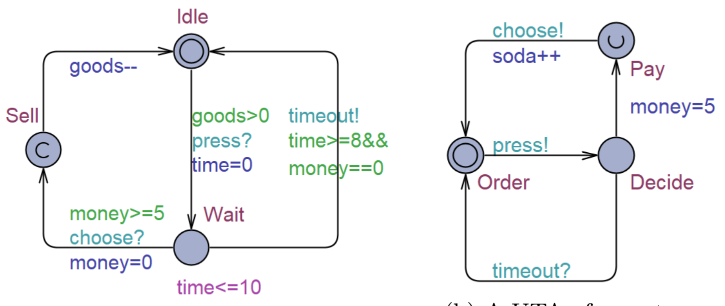
*where $L$ is a finite set of locations, $l_0$ is the initial location, $X$ is a finite set of non-negative real-valued clocks, $\Sigma$ is a finite set of actions, $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is a finite set of edges, where $\mathcal{B}(X)$ is the set of guards over $X$, that is, conjunctive formulas of clock constraints of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in X$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, $2^X$ is a set of clocks in $X$ that are reset, and $Inv : L \to \mathcal{B}(X)$ is a partial function assigning invariants to locations.* □

The semantics of a TA $\mathcal{A}$ is defined as a *timed transition system* over states $(l, v)$, where $l$ is a location and $v \in \mathbb{R}^X$ represents the valuation of the clocks in that state, with the initial state $s_0 = (l_0, v_0)$, where $v_0$ assigns all clocks in $X$ to zero. There are two kinds of transitions:

1. *delay transitions*: $(l, v) \xrightarrow{d} (l, v \oplus d)$, where $v \oplus d$ is the result obtained by incrementing all clocks of the automaton with the delay amount $d \in \mathbb{R}^+$ such that $v \oplus d \models I(l)$, and

2. *discrete transitions*: $(l, v) \xrightarrow{a} (l', v')$, corresponding to traversing an edge $l \xrightarrow{g,a,r} l'$ for which the guard $g$ evaluates to *true* in the source state $(l, v)$, $a \in \Sigma$ is an action, $r$ is a set of clocks that are reset over the edge, and clock valuation $v'$ of the target state $(l', v')$ is obtained from $v$ by resetting all clocks in $r$ such that $v' \models Inv(l')$.

UPPAAL [9] is a state-of-the-art model checker for real-time systems. It supports modeling, simulation, and model checking, and uses an extension of TA with data variables, synchronization channels, urgent and committed locations etc., as the modeling formalism, which we call UPPAAL TA (UTA). In UPPAAL, UTA can be composed in parallel as a *network* of UTA synchronized via *channels* (an edge decorated with channel $a!$ is synchronized with one decorated with $a?$ by handshake). An example of UPPAAL model is depicted in Figure 10.2, consisting of two automata: a customer ordering sodas and an impatient vending machine kicking its customers out of the system when they do not order quickly enough (i.e., 10 time units is the maximum waiting time). A UTA of the



(a) A UTA of a vending machine

(b) A UTA of a customer

Figure 10.2: An example of UTA modeling a customer ordering food from an impatient vending machine.

vending machine shown in Figure 10.2a has 3 locations named `Idle`, `Sell`, and `Wait`. *Edges* are the direct lines that connect *locations*, which can be decorated by *guards*, *channels*, and *updates*. A clock variable `time` measures the elapse of time, and is used in the *invariants* on locations (e.g., `time`<=`10`) and in the *guards* on edges (e.g., `time`>=`8`). In UTA, locations can be labeled as *urgent*, denoted by encircled ∪, which forbids delaying in the locations (e.g., `Pay` of the customer TA in Figure 10.2b), or *committed*, denoted by encircled $C$, which not only forbid time from elapsing but also require the network of UTA to transit the next edge from one of the committed locations. Such an example of committed locations is *location* `Sell` of the vendor UTA in Figure 10.2b. UTA also extends TA with data variables (integer and Boolean variables), which can be updated via C-code functions or assignments on edges. For example, on the *edge* from *locations* `Decide` to `Pay` of the customer TA, an integer variable `money` is updated to 5, meaning that the customer has paid 5 Swedish crowns to the vending machine.

UPPAAL can verify properties formalized as queries in a subset of *Timed Computation Tree Logic* (TCTL) [12]. Given an atomic proposition $p$ over the locations, clocks, and data variables of the UTA, the UPPAAL queries that are used in this paper are: (i) **Invariance**: $A$`[]` $p$ means that for all paths, for all states in each path, $p$ is satisfied, (ii) **Liveness**: $A$`<>` $p$ means that for all paths, $p$ is satisfied by at least one state in each path, (iii) **Reachability**: $E$`<>` $p$ means that there exists a path where $p$ is satisfied by at least one state of the path, and (iv)

**Timed-bounded Reachability**: $E{<>}_{\leq T}\ p$ means that there exists a path where $p$ is satisfied by at least one state of the path within $T$ time units.

### 10.3.2 Devices for Assisted Living (DALi)

The DALi algorithm has been proposed previously [6], and is designed to provide motion planning in complex environments. The planner takes into account environmental constraints and user's preferences.

DALi consists of two parts: long-term and short-term planners. The former one is performed prior to actual navigation and searches for paths accounting for area topology, user goals, and foreseen obstacles or problems along the way. During navigation unforeseen obstacles can appear, e.g. a group of people obstructs the path, its detection starts a short term planner attempting to find a minimal deviation from the path that preserves all the constraints. Short term planning is often required to operate on low resources and to provide a quick response, therefore long term planner cannot be used for quick search for path deviations.

In this paper, we are interested in a long term planner of DALi. At the first step the algorithm transforms the area into a graph with a sufficient granularity to represent paths between points. Permanent obstacles are excluded from the graph. Other environmental constraints and user preferences are stored within nodes and edges of the graph with one of the following options:

- Soft constraints taken from users' preferences and indicating zones that the planner would try to visit or to avoid. Each soft constraint is characterized by a triple (*center, radius, intensity*): it affects paths within the *radius* from its *center* and its intensity is the highest at the *center* decreasing with the distance from the *center*. A proposed path is deviated towards or outwards of the center within the radius based on the intensity level.

- A heat map of the environment indicating areas with high occupancy; navigation through such areas is slower by a specified factor.

- Anomalies or temporary obstacles that are areas inaccessible during certain periods of time. The long-term planner considers foreknown anomalies, assuming that their appearance and disappearance time is provided.

The long-term planner of DALI is based on Dijkstra's shortest path algorithm [13]. The algorithm maintains a set of nodes to which the shortest distances from a source node are computed. Starting from the source node, at each step the algorithm selects a new node that has the shortest distance from the source node and adds it to the set. DALI modifies Dijkstra's algorithm by taking into account environmental constraints and users' preferences during the choices of the nodes to be added into the set. Temporary obstacles prevent selection of the nodes inside these obstacles during the inaccessibility time. Heat maps affect the actual lengths of the edges. Soft constraints add virtual coefficients to the distances inside the zones covered by these constraints. Note that soft constraints do not affect the real travel time (the coefficient is virtual), thus DALI might return a path that is not the shortest. The coefficients of the soft constraints bound the extra length of the returned path, that is, if a detour for satisfying a soft constrain is too long to satisfy the time limit of reaching the destination, the constraint would be ignored.

## 10.4 Mission Planning Methodology

In this section, we introduce our methodology for automated mission planning for multiple agents. We describe the overall procedure followed by the detailed description of each step.

Mission planning is composed of two main aspects: task scheduling and path planning. The former defines which tasks, in which order, at what time, and by which agent should be executed. The latter indicates the traveling path between milestones. Note the dependency of task scheduling on path planning: the knowledge of the traveling time is needed for scheduling.

We propose a *UTA-based mission planner* for agents. The path-planning aspect is based on an adapted version of the DALI algorithm. The DALI-based path planner takes the information of the map, including the navigation area, special areas (e.g., forbidden areas), milestones, tasks, and agents, and computes paths connecting milestones to each other regardless of the visiting order. The task-scheduling aspect is an adapted version of TAMAA (Timed-Automata based Mission Planner for Autonomous Agents) [7]. The TAMAA-based task scheduler employs UPPAALto synthesize an optimal schedule satisfying all the task constraints such as the correct order of task execution. In general, task

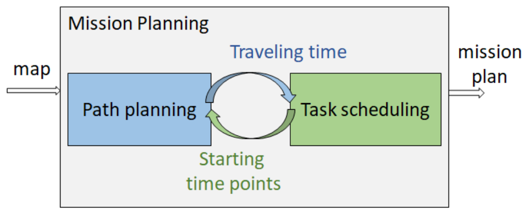


Figure 10.3: Iterative process of mission planning

scheduling sets up the skeleton of the mission plan, which orders the actions of movement and task execution, and path planning fills in the generated concrete routes between every pair of milestones. While theoretically it is possible to employ UPPAALfor both path planning and task scheduling, in practice full mission planning in UPPAALis infeasible due to the scalability problem, and the separation into two aspects is designed to simplify the computations [7].

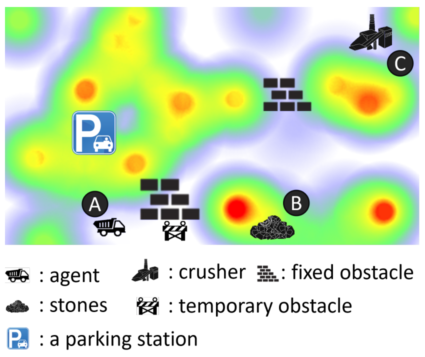
To ensure the correct mission planning, path planning and task scheduling have to interact with each other: task scheduling requires the traveling time between milestones while path planning needs to know when the trip would take place in order to generate paths avoiding temporary obstacles. Therefore, both parts are executed in a loop until all constraints are satisfied by the resulting mission plan. The overall workflow consists of the following steps:

1. Mission planning receives input information about the navigation area and its abnormalities, as well as required tasks and their constraints.

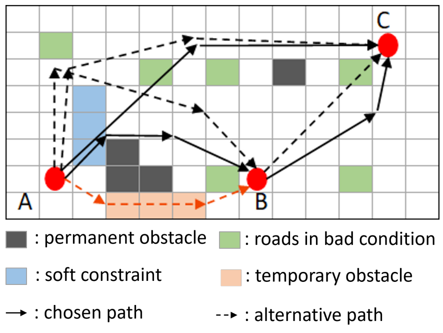
2. Path planning calculates potential paths and traveling time for agents between every pair of milestones.

3. TAMAA builds UTA models that are verified in UPPAAL. In case of successful verification, UPPAAL provides execution traces of the models that satisfy all task constraints. Subsequently, a task schedule is generated from the traces.

4. Path planning checks whether the scheduled travels cross temporary obstacles when they are active. If a conflict is found, the planning returns to step 2 where the affected paths are updated.



(a) An example of the autonomous quarry



(b) A grid of the example

Figure 10.4: An example of the autonomous quarry and a grid that discretizes the environment.

5. If both task and path constraints are satisfied, the iteration ends and a resulting mission plan is returned.

Figure 10.3 illustrates our solution. In the following subsections, we provide detailed descriptions of the path planner (DALI) and of the task scheduler (TAMAA), as well as their integration. To illustrate the following models and algorithms, we use a running example based on the case study described in Section 10.2. Figure 10.4a depicts a small quarry, where an agent (i.e., an autonomous truck) originally located at milestone $A$ should load stones at milestone $B$ and deliver them to a crusher at milestone $C$. Two stationary obstacles are located in the quarry (brick walls in the figure). The area next to the stationary obstacle at the bottom is temporarily blocked. A parking station is an area that is recommended to be avoided. Moreover, road conditions are poor and the color scheme indicates the speed reduction (white areas can be passed at full speed while red areas slow down the agents the most).

### 10.4.1 Improved DALi for Path Planning

The path planning algorithm is utilized to compute paths between all milestones. The traveling time of the paths is used by TAMAA for task scheduling. Note that not all paths are included in the final mission plan: only travels scheduled by TAMAA would be used. In the running example in Figure 10.4a, the path planning algorithm computes paths between every pair of milestones, three in total (i.e., A to B, B to C, and A to C), yet the final mission plan would use only two of them (i.e., A to B and B to C).

Path planning has to be capable of providing navigation in complex environments, considering obstacles, road conditions, and users' additional preferences. The DALi algorithm supports different types of environmental constraints, thus we select it for our methodology. To adapt DALi to our use case, we transform DALi environmental constraints to be applicable in a quarry. In addition, we tune the algorithm with several optimizations as shown below.

The preliminary step of the DALi algorithm is a transformation of the navigation area into a graph and annotation of graph elements with environmental conditions. Figure 10.4b illustrates a discretization of the area with a Cartesian grid where each cell represents a node in the graph and neighbour cells are connected by edges. Each edge has a length equal to the distance between centers of cells connected by the edge. The initial positions of agents as well as milestones are assigned to the nodes corresponding to the cells that they are located at, respectively. Environmental constraints are encoded into the graph as follows.

- Permanent obstacles or areas that are always impassable (dark grey in Figure 10.4b) are excluded from the graph. There are no nodes corresponding to such areas and no edges connecting them.
- Temporary obstacles or areas that are impassable for a specified time interval (light orange in Figure 10.4b) are kept in the graph

unlike permanent obstacles. Nodes in such areas are annotated with periods of inaccessibility that would be used by the path planning.

- Areas with bad road conditions are specified with a heat map used in DALi (green in Figure 10.4b). In the rest of the paper we refer to such areas as *heat areas*. Agents in a heat area have to reduce their respective speed by a given factor that is stored within the edges connecting nodes inside the area.

- Soft constraints marking some areas as "undesirable" (blue in Figure 10.4b) are defined differently from the soft constraints in DALi. Agents are allowed to pass through such areas however a virtual coefficient is added to lengths of edges making them less preferential to the path planner. Different from the original DALi where the coefficient depends on the distance to the area center, we consider the uniform coefficient in the whole undesirable area. In this way, we ensure that traversing the undesirable area even close to its boundary is discouraged. Note that, contrary to the heat areas that also affect the edges' lengths, this coefficient is virtual and only applied during the path finding but does not influence the actual traveling time on the paths.

The core step of the algorithm is a computation of the path between each pair of milestones on the created graph. The original DALi is a single-source single-target algorithm, i.e., it computes a path between a *source* and a *target*, and is based on Dijkstra's shortest path algorithm [13]. Algorithm 2 is called for each pair of milestones computing the shortest path between them while considering the environmental constraints. Each node stores a distance to the *source*, initially infinite. A priority queue orders nodes by their current distances to the *source* (line 12). The main loop (lines 18-29) takes a node with the smallest distance and updates the distances from all its neighbours to the *source*. The update ensures the avoidance of temporary obstacles (lines 21 - 22) and calculates the distances taking in account soft constraints and heat areas (line 23). The algorithm terminates when the distance to the *target* node is computed and the path is obtained by moving in the graph via references to the previous nodes (line 30).

**Algorithm 2:** Path Planning Algorithm

```
class Node
    Edge edges[]                                          // Outgoing edges
    Double distance, vDistance  // Distance to source (virtual - with soft constraints)
    Node previous                                 // Previous node on the path
    Double softConstraint   // Soft constraint coefficient; 1 if no constraint
    TimeInterval temporaryObstacle       // Inaccessibility time if inside a temporary obstacle
class Edge
    Node start, end
    Double length
    Double heat     // heat map speed reduction factor; value ∈ $[0, 1)$
Function FindPath(Node[] area, Node source, Node target, Agent agent, Double startTime, Bool noTemp, Bool softExist)
    PriorityQueue queue
    foreach Node node : area do
        $node.distance := inf$
        $queue.add(node)$
    $source.distance := 0$
    Node processed[]
    while $queue \mathrel{!}= \emptyset$ && $target \notin processed$ do
        Node $currentNode := queue.remove()$
        foreach Edge edge : currentNode.edges do
            if $noTemp \,||\, (edge.end \in processed \,||\, IsInsideTemporaryObstacle(edge, agent, startTime))$ then
                skip
            Double $newDistance := currentNode.distance + softExist \times edge.length \times edge.end.softConstraint/(1 - edge.heat)$
            if $edge.end.vDistance > newDistance$ then
                $edge.end.vDistance := newDistance$
                $edge.end.previous := currentNode$
                $edge.end.distance := currentNode.distance + edge.length/(1 - edge.heat)$
            $queue.add(edge.end)$
        $processed.add(currentNode)$
    return $ExtractPath(target)$    // traversal from target to source via node.previous
Function IsInsideTemporaryObstacle(Edge edge, Agent agent, Double startTime)
    if $edge.end.temporaryObstacle \neq null$ then
        Double $time := startTime + (edge.length + edge.start.distance)/agent.speed$
        return $time \in edge.end.temporaryObstacle$
    return False
```

Within this work we propose two optimizations of the original DALI algorithm. The first optimization takes into account that our method requires to compute paths between every pair of milestones and that the Dijkstra's algorithm on which DALI is based can be modified into single-source multiple-target path planner. The extension of DALI is straightforward: the algorithm continues the main loop until the distances between every pair of milestones are computed. This optimization allows to reduce the number of calls to the path planner.

The second extension is inspired by the heuristics from the A* algorithm [14]. Whenever the algorithm updates the distance of a node (line 23), it considers not only the distance to the *source* but also a distance estimation to the *target*. In particular, each node stores an estimation of the entire path length from the *source* to the *target* passing through the node and the priority queue sorts the nodes based on the estimation. For a node $n$ the estimation is computed as $n.distance + dist(n,\ target)$, where *dist* returns the Euclidean distance. Such heuristic guides the exploration of the graph towards the *target* and, in general, finds the shortest paths faster, especially in areas with few obstacles. In the remainer of the paper we call the DALI extension with the A* heuristic as DALI*. Note that the two extensions above are incompatible: an A* heuristic assumes that the algorithm is single-source single-target. The selection between the two optimizations must take into account the number of milestones, which affects the number of paths to compute, and the size of the Cartesian grid that discretizes the environment (Fig. 10.4b).

At the second step of the overall workflow (see the beginning of Section 10.4), a path planning algorithm computes the path between every pair of milestones. Since the path planner does not know at which time a path would be used, temporary obstacles are not considered during this step, i.e., the Boolean variable $noTemp$ on line 21 is true. In our running example in Figure 10.4b, the red path between A and B through the temporary obstacle would be selected. A path between B and C would not be direct: a heat area on the way would significantly slow down the agent and make a deviation faster. The path between A and C is also deviated due to the soft constraint, yet such path is far from optimal and may affect the timing constraints of the mission plan.

Temporary obstacles are considered during step 4 of the overall workflow (see the beginning of Section 10.4). At this point the selected paths and their starting time points are known, thus making it possible to

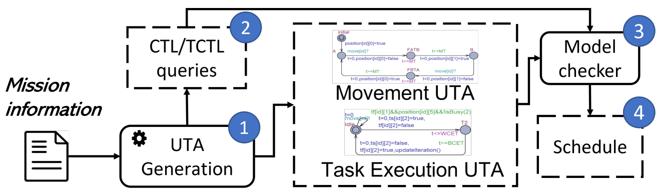


Figure 10.5: Overview of the workflow of model generation and mission plan synthesis in TAMAA

check whether such obstacles have been passed at the wrong time when the temporary obstacles exist. If that is the case, such paths are recomputed with a consideration of the temporary obstacles (i.e., $noTemp$ on line 21 is now false). In the running example (see Fig. 10.4b), the path between A and B is recomputed due to the temporary obstacle and a new and longer path avoiding both the temporary obstacle and the area with a soft constraint is computed. The new path and its length is then given to TAMAA for model generation.

It might be the case that paths avoiding areas with soft constraints are too long, which causes the timing constraint of a global mission to be violated. In this case, paths are recomputed by the path planner with soft constraints ineffective (i.e., `softExist` is false, which is converted to 0 at line 23). This recomputation may improve some paths and the updated traveling time is passed to TAMAA for task rescheduling. In the running example, such recomputation would output yet another path between A and B.

### 10.4.2 TAMAA for Task Scheduling

Now, we recall the mission-related requirements of agents, mentioned in Section 10.2, which agents need to fulfill. To ensure the correct work of the agents, we need a method that guarantees to meet all the desired requirements, including correct task scheduling. Model checking can provide such guarantees given a modeling formalism for agents and constraints. We propose a method named *TAMAA(Timed-Automata-based Mission planner for Autonomous Agants)* that is able to automatically generate models and synthesize task schedules that satisfy the formalized requirements by using model checking. TAMAA employs UPPAAL as the model checker that uses the UPPAAL timed automata (UTA) formalism for modeling timed behavior of agents and (Timed) Computation Tree Language ((T)CTL) for requirement specification.

Figure 10.5 depicts the workflow of our approach:

1. *UTA generation*: mission information, e.g., topology of the map, and information about the agents and their tasks, are input into the model-generation module, where a set of UTA that models the agents' movement and task execution are generated automatically.

2. *Query generation*: (T)CTL queries for synthesizing schedules are generated automatically. Based on our templates of queries, users can manually modify the queries according to their own requirements.

3. *Trace generation*: The UTA models are verified with UPPAAL against the (T)CTL queries. If the model checker finds an execution trace of the model that meets all the requirements, the trace is returned; otherwise, a verdict that the query is not satisfied is returned.

4. *Schedule generation*: The returned trace is parsed and a schedule of actions (i.e., movement and task execution) is generated based on the trace.

The method automatically generates models, formalizes requirements, and synthesizes traces. The TAMAA method's automation of the process simplifies its application: no user action is required after setting up the tasks, milestones, and navigation area. Nevertheless, we leave the possibility to modify models and to add requirements so that our method can be used in applications with individual needs that are not expressible with the existing models. We introduce this in detail in Section 10.7. In addition, as the traces are generated by exhaustive traversal of the model state space, we can select the fastest ones that finish the tasks while holding the other requirements, e.g., *milestone matching*. In the following, we introduce the theoretical and technical details of TAMAA including formal definitions of the concepts, two algorithms for model generation, and the templates of queries that formalize the requirements presented in Section 10.2.

### Definitions of Concepts

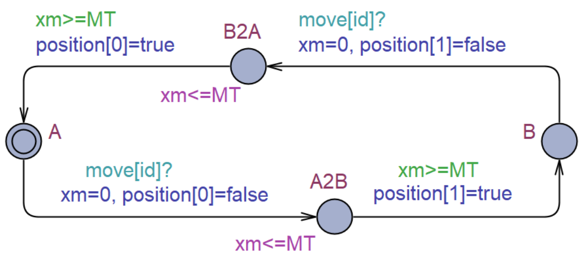
In our approach, autonomous agents are characterized by their speeds and a set of tasks that they are supposed to execute for accomplishing the entire mission. The environment where agents work in contains a number of milestones where the tasks are supposed to be carried out. Therefore, agents with a certain subset of tasks should visit the right milestones. To accomplish the mission, there are two types of actions that agents can perform, namely *moving* and *executing* tasks. Therefore, we split the agent model into two UTA, one taking care of the movement and one of the task execution.

**Definition 10** (Agent's Movement UTA)**.** *Given an autonomous agent AA, the movement of AA is defined as a* UTA *in the following form:*

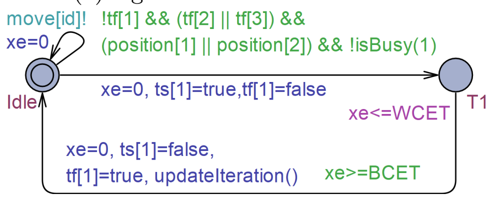
$$MV = < P_m, p_0, U_m, \Sigma_m, E_m, I_m >, \tag{10.4}$$

*where:*

- $P_m = P_m^s \cup P_m^t$ *is a finite set of locations, where* $P_m^s$ *represents the set of milestones in the environment, and* $P_m^t$ *is designed for measuring the traveling time between milestones;*
- $p_0 \in P_m^s$ *is an initial location representing the milestone where the agent is initially positioned;*
- $U_m = \{x_m,\ position\}$*, where* $x_m$ *is a clock variable for measuring the traveling time, and "position", which is shared with other* UTA*, is an array of Boolean variables that stores whether the agent is at a milestone or not;*
- $\Sigma_m = \{move, \tau\}$ *is a set of channels, where "move" models the synchronization with task execution automaton described below in Definition 12, and* $\tau$ *denotes internal or empty actions without synchronization;*
- $E_m = E_m^c \cup E_m^u$ *is a finite set of edges connecting locations, where* $E_m^c \subseteq P_m^s \times \{true\} \times \{move\} \times F_m \times P_m^t$*, where* $F_m$ *is a set of functions that update the value of the Boolean array "position" and reset the clock* $x_m$*, and* $E_m^u \subseteq P_m^t \times B_m(x_m) \times F_m \times P_m^s$*, where* $B_m(x_m)$ *is a set of guards containing clock constraints of the form* $x_m \geq \delta$*, where* $\delta \in \mathbb{R}_{\geq 0}$ *is the traveling time between two milestones;*



(a) Agent movement as an UTA


move[id]!
xe=0
!tf[1] && (tf[2] || tf[3]) &&
(position[1] || position[2]) && !isBusy(1)
xe=0, ts[1]=true,tf[1]=false
Idle
xe<=WCET
T1
xe=0, ts[1]=false,
tf[1]=true, updateIteration()
xe>=BCET

(b) Agent execution of tasks as an UTA

Figure 10.6: Examples of agent movement UTA and task-execution UTA.

- $I_m : P_m^t \to B_t(x_m)$ *is a function that assigns invariants to locations in* $P_m^t$*, where* $B_t(x_m)$ *contains clock constraints of the form* $x_m \leq \delta$*, where* $\delta \in \mathbb{R}_{\geq 0}$ *is the traveling time between two milestones.*

Figure 10.6a illustrates the movement UTA for two milestones $A$ and $B$. The UTAhas four *locations*: two *milestone* locations representing the milestones, i.e., $P_m^s = \{A, B\}$, and two *traveling* locations representing intermediate positions necessary to capture traveling between milestones, i.e., $P_m^t = \{A2B, B2A\}$. Edges in $E_m^c$ that connect the milestone locations with the traveling locations are labeled with a *channel* `move`, in our example, the edges from *locations* `A` to `A2B` and from *locations* `B` to `B2A`, respectively. The *channel* `move` is for synchronizing the movement UTA with the task execution UTA when the latter informs that the movement has started. A second group of edges (i.e., $E_m^u$) models the arrival to the destination. To model the traveling time we use invariants of the form $x_m \leq$ `MT` and guards in the form of $x_m \geq$ `MT`. The clock variable $x_m$ is for measuring the traveling time between two milestones. The set $F_m$ in this UTA contains the assignments that reset the clock $x_m$ and modify a specific element in the Boolean array `position`. We use the array `position` to keep track of current position of an agent, which can be accessed by other UTA.

The second part of an agent model formalizes task execution. First, we define tasks as follows:

**Definition 11** (Task)**.** *A task is defined as a tuple, as follows:*

$$TS = (\texttt{BCET}, \texttt{WCET}, \texttt{isStarted}, \texttt{isFinished}, \texttt{Var}, \texttt{pre}, \texttt{Mil}), \quad (10.5)$$

*where:*

- `BCET` *is the best case execution time;*
- `WCET` *is the worst case execution time;*
- `isStarted` *is a Boolean variable denoting if the task has started;*
- `isFinished` *is a Boolean variable denoting if the task has finished;*
- `Var` *is a set of functions that update the variables in the task-execution UTA (Definition 12) after the task finishes;*
- `pre` *is a precondition that must be met to start the task, which can take into account execution status of other tasks and global variables. Formally,* $\texttt{pre} = p \mid \neg\texttt{pre} \mid \texttt{pre} \vee \texttt{pre} \mid \texttt{pre} \wedge \texttt{pre}$, *where* $p$ *is an atomic proposition over* $\mathcal{S} \cup \mathcal{F} \cup \mathcal{V}\dashv\nabla$, *where* $\mathcal{S}$, $\mathcal{F}$, *and* $\mathcal{V}\dashv\nabla$ *are three sets consisting of Boolean variables* `isStarted`, `isFinished`, *and the variables updated in* `Var` *of all tasks, respectively;*
- `Mil` *is a set of milestones where the task is allowed to be executed. A task can be executed at multiple milestones.*

Table 10.1 shows two examples of tasks, which we use to illustrate Definition 11.

Table 10.1: An example of tasks for the autonomous trucks in Figure 10.4.

| | BCET (mins) | WCET (mins) | isStarted | isFinished | Var | pre | Mil |
|---|---|---|---|---|---|---|---|
| Loading | 5 | 10 | false | false | full | / | B |
| Unloading | 8 | 14 | false | false | unload(stone) | Loading.isFinished | C |

Initially, an agent starts to load stones near the stone piles at milestone *B* (*loading.isStarted* turns *true*). When the loading task is finished (*Loading.isFinished* turns *true*, while *Loading.Started* turns *false*), a Boolean variable *full* indicating whether the agent is fully loaded is set to *true*. At the next step, the agents are supposed to unload the stones into the primary crusher at milestone $C$. Therefore, the unloading task has a precondition *loading.isFinished*. Note that preconditions can be much more complex Boolean expressions, e.g., $A.pre = (B.isFinished \mid\mid \neg C.isStarted)\ \&\ D.isStarted$, which means task $A$ can start only when task $B$ is done or task $C$ has not started, and after task $D$ has started. When the unloading task is done, an integer named *stone* is increased, which indicates the amount of crushed stones.

The execution time of a task is usually specified with a time interval (i.e., [`BCET`, `WCET`]). In a 1-player game [15], the environment is under the total control of agents, which means agents can choose any time point within the time interval to finish the task. The goal of task scheduling in this paper is to find the schedules that finish tasks in the quickest way. Notably, the quickest schedules do not necessarily mean always using BCET for all tasks. In some cases, prolonging the execution time of some tasks can be more efficient for the entire mission, especially when multiple agents are working in the same environment. Additionally, some applications require the tasks to be executed multiple times before the final goal is reached. Like the example in Table 10.1, autonomous trucks are asked to repeat the trip of loading and unloading until all the stones are transferred to the primary crusher (i.e., variable *stone* is zero), and the tasks should be executed once and only once during one trip, i.e., before all tasks are completed. With all the requirements, how to assign starting time and ending time of tasks to each of the agents is the question that is answered by task scheduling. Now, we define the task execution UTA based on the definition of tasks.

**Definition 12** (Task Execution UTA). *Given an agent AA and a set of tasks $\mathcal{T}$, task execution of AA is defined as a UTA of the form:*

$$TE = (N_e, n_0, U_e, \Sigma_e, E_e, I_e), \tag{10.6}$$

*where:*

- $N_e = \{n_0\} \cup N_e^t$ *is a set of locations, where* $N_e^t = \{n_t \mid t \in \mathcal{T}\}$*;*
- $n_0 \in N_e$ *is the initial location, which stands for the idle status of AA;*
- $U_e = \{x_e, t_s, t_f, ite\}$*, where $x_e$ is a clock that is reset whenever a task finishes, $t_s$ and $t_f$ are Boolean arrays that store the statuses*

*of tasks (isStarted and isFinished in Definition 11, respectively), and ite* $\in \mathbb{N}$ *stores the current iterations of all tasks. At the end of each iteration, ite is incremented by 1, while* $t_s$ *and* $t_f$ *are reset to false for a new round of tasks execution;*

- $\Sigma_e = \{move, \tau\}$ *is a set of channels;*
- $E_e = E_i^d \cup E_e^d \cup \{e_s\}$*, where* $E_i^d \subseteq \{n_0\} \times \{\tau\} \times B_e^i(U_e) \times F_e \times N_e^t$ *is a set of edges from* $n_0$ *to* $n_t \in N_e^t$*,* $F_e$ *is a set of functions updating the variables in* $U_e$*,* $B_e^i(U_e)$ *is a set of guards consisting of the preconditions and milestone requirements of tasks,* $E_e^d \subseteq N_e^t \times \{\tau\} \times B_e^d(x_e) \times F_e \times \{n_0\}$ *is a set of edges from* $n_t \in N_e^t$ *to* $n_0$*,* $B_e^d(x_e)$ *is a set of guards containing clock constraints of the form* $x_e \geq BCET$ *of a task,* $e_s$ *is a self-loop edge on* $n_0$ *that is labeled with channel move and an assignment* $x_e = 0$*;*
- $I_e : N_e^t \to B_i(x_e)$ *is a function assigning invariants to locations in* $N_e^t$*. The invariants are of form* $x_e \leq WCET$ *of a task.*

An example of the task execution UTA is depicted in Figure 10.6b, where *location* `Idle` is the initial *location* $n_0$, and *location* `T1` represents a task. The *channel* `move` labels the self-loop *edge* on *location* `Idle`, which synchronizes the movement UTA and the task execution UTA. Due to the synchronization, movement cannot start during task execution but only when the agent is idle. *Location* `T1` and its outgoing *edge* are labeled with an invariant (`t<=WCET`) and a *guard* (`t>=BCET`), respectively, to ensure the execution time within *BCET* and *WCET*. Function `updateIteration` on the edge from `T1` to `Idle` is for incrementing the variable *ite* when an agent finishes all its tasks. For a specific case, additional functions can be added to this edge too, for example, the function `unload(stone)` in Table 10.1. Moreover, the guard `!tf[1]` forbids the multiple execution of the task during a single round of tasks iteration; the next execution of this task can be done only in the next iteration after the reset of the arrays `ts` and `tf`. The guard on the edge from *Idle* to `T1` presents the precondition (i.e., `!tf[1]` && (`tf[2]` || `tf[3]`)), the milestone requirement (i.e., `position[1]` || `position[2]`), and the mutual-exclusive requirement of task *T1* (i.e., `!isBusy(1)`). The function `isBusy(1)` checks if `T1` is being executed by other agents or not. As preconditions defined in Definition 11, the function `isBusy(1)` uses the "isStarted" and "isFinished" variables of the same task executed by other agents to see if task *T1* is being executed.

**Algorithm 3:** Movement UTA Generation

```
Function CreateMovementUTA(int[][] map, int speed)
    UTA movementUTA
    int $i := 0,\ j := 0$
    for $i < map.size$ do
        Location $l_i := createLocation(“L_i”)$   // Create a location with the name $L_i$
        $movementUTA.addLocation(l_i)$
        $i++$
    $i := 0$
    for $i < map.size$ do
        Location $l_i := movementUTA.getLocation(“L_i”)$   // Get a location with the name $L_i$
        for $j < map[i].size$ && $i \neq j$ do
            Location $l_j := movementUTA.getLocation(“L_j”)$   // Get a location with the name $L_j$
            Location $l_{FiTj} := createLocation(“F_iT_j”)$   // Create a location with the name $F_iT_j$
            $l_{FiTj}.invariant = “x_m \leq map[i][j]/speed”$
            $movementUTA.addLocation(l_{FiTj})$
            Edge $e_1 := createEdge(l_i, l_{FiTj})$   // Create an edge from $l_i$ to $l_{FiTj}$
            $e_1.channel := “move?”$
            $e_1.assignments := “x_m = 0, position[i] = false”$
            $movementUTA.addEdge(e_1)$
            Edge $e_2 := createEdge(l_{FiTj}, l_j)$   // Create an edge from $l_{FiTj}$ to $l_j$
            $e_2.guard := “x_m \geq map[i][j]/speed”$
            $e_2.assignments := “x_m = 0, position[j] = true”$
            $movementUTA.addEdge(e_2)$
            $j++$
        $i++$
    return movementUTA
```

**Algorithm 4:** Task-execution UTA Generation

```
Function CreateTaskUTA(int agentID, TS tasks[])
    UTA $taskexeUTA$
    Location $Idle := createLocation(\text{“}Idle\text{”})$   // Create a location with the name $Idle$
    $taskexeUTA.addLocation(Idle)$
    $e_0 := createEdge(Idle, Idle)$   // Create a self-loop edge of location $Idle$
    $e_0.channel :=$ “$move[agentID]!$”
    $e_0.assignment :=$ “$x_e = 0$”
    $taskexeUTA.addEdge(e_0)$
    for $i < tasks.size$ do
        Location $T_i := createLocation(T_i)$   // Create a location with the name $T_i$
        $T_i.invariant := x_e \leq tasks[i].WCET$
        $taskexeUTA.addLocation(T_i)$
        Edge $e_1 := createEdge(Idle, T_i)$   // Create an edge from $Idle$ to $T_i$
        $e_1.guard :=$
          “$\neg tf[i] \land tasks[i].Pre \land (\bigvee_{l \in tasks[i].Mil} position[l]) \land !isBusy(i)$”
        $e_1.assignments :=$ “$x_: = 0,\ ts[i] = true,\ tf[i] = false$”
        $taskexeUTA.addEdge(e_1)$
        Edge $e_2 := createEdge(T_i, Idle)$   // Create an edge from $T_i$ to $Idle$
        $e_2.guard :=$ “$x_e \geq tasks[i].BCET$”
        $e_2.assignment :=$ “$x_e = 0,\ ts[i] = false,\ tf[i] :=$
          $true,\ updateIteration()$”
        $taskexeUTA.addEdge(e_2)$
        $i++$
    return $taskexeUTA$
```

Definitions 10 and 12 define the models of agent movement and task execution, respectively. A network of these UTA models the behavior of multiple agents working collectively to reach a common goal respecting a given set of constraints, such as mutual-exclusiveness of tasks. The definitions present the foundation for describing multi-agent systems with a formal modeling language. Next, we introduce how the course of modeling is automated by two algorithms.

### Generation of UTA

In this section, we introduce the algorithms generating the movement and task execution UTA. Figure 10.6 illustrates the results of algorithms application. Movement UTA (Definition 10) of an agent is generated by

Algorithm 3. Its inputs are a two-dimensional array called *map*, which stores the distance between every pair of milestones, and a variable called *speed*, which is the agent's speed. The algorithm starts by creating a location for each milestone (lines 4 - 7). Next, traveling between each pair of milestones is represented by a *traveling location* (lines 13 - 14) and two edges connecting the milestones via the traveling *location* (lines 16 - 23). *Guards*, *channels*, and *invariants* are assigned to location and edges according to Definition 10.

Algorithm 4 describes the generation of an agent's task execution UTA. Lines 3 - 8 create a *location* to represent the idle status of the agent and a self loop at this *location* to synchronize with the agent's movement UTA. This is a single point of synchronization between the two automata since task execution and movement are mutually exclusive. Lines 10 - 20 create *locations* representing the execution of each task and edges that connect these *locations* with the `idle` *location*. Line 14 assigns a *guard* to the *edges* coming from the *location* `Idle`. The *guard* regulates the UTA to start the task only when the task's precondition holds ($tasks[i].Pre$) and the agent is positioned at one of the right milestones $((\bigvee_{l \in tasks[i].Mil} position[l]))$, where *position* is an array whose values are changed by the movement UTA of the agent. The guard $\neg tf[i]$ means that the task is not finished yet in this round of task iteration. The function $updateIteration()$ in line 19 increments the variable *ite* belonging to this UTA if all tasks of the corresponding agent are finished, and turns the variables in $tf$ and $ts$ to $false$, indicating a new round of iteration is about to start.

### Formalizing Requirements as UPPAAL Queries

In Section 10.2, we provide a list of typical requirements of our use case. In this section we describe how they are formalized for the model checking. Queries for verification of each agent are created based on the templates presented below (formulas (10.7) to (10.10)). We use index '`a`' to denote an agent in queries. We use $\texttt{task}_a$ (respectively, $\texttt{move}_a$) to denote the task execution UTA (respectively, movement UTA) of an agent `a`, and `Ti` (respectively, `Pi`) to denote any *location* in the task execution UTA (respectively, movement UTA). A clock variable `x` is used to measure the global time. Two Boolean arrays named `ts` and `tf` indicate whether tasks have been started and finished, respectively. Two constant integers `ALL` and `LIMIT` denote the requested number of iterations of tasks and the time constraint to accomplish the entire mission, respectively.

- *Milestone matching*: Agents must be located at the right milestone while executing a task. Assuming task $T_i$ must be carried out at one of the milestones: $P_i$, $P_{i+1}$, ..., $P_k$, the following queries are checked for each agent `a`:

  $$\texttt{E<> ts}_a\texttt{[i]} \tag{10.7}$$

  $$\texttt{A[] task}_a\texttt{.Ti imply (move}_a\texttt{.Pi} \mid\mid \texttt{...} \mid\mid \texttt{move}_a\texttt{.Pk)} \tag{10.8}$$

  Query (10.7) is for verifying whether task $T_i$ ever starts, after which Query (10.8) checks whether task $T_i$ is carried out at the right milestone.

- *Task sequence*: Tasks must eventually be executed, and to start the execution, their preconditions must be satisfied. Assuming task $T_i$ can start only after task $T_j$ finishes, and at the beginning of each task iteration, all elements in both `ts` and `tf` are set to *false*, the corresponding queries are designed:

  $$\texttt{A[] ts}_a\texttt{[i] imply tf}_a\texttt{[j]} \tag{10.9}$$

  The first part of the requirement, i.e., whether task $T_i$ ever starts, is verified in the requirement of *milestone matching* by checking Query (10.7). Query (10.9) checks the second part of the requirement: task $T_i$ never starts before the required preceding task $T_j$ has finished.

- *Timing*: Tasks must be finished within a time frame in order to maintain a certain level of productivity. The following query is used to capture this requirement:

  $$\texttt{E<> iteration[a]>=ALL and gClock <= LIMIT}, \tag{10.10}$$

  where *gClock* is a global clock that is not reset by any transition. Query (10.10) checks the reachability of a state where all tasks are executed for `ALL` rounds within `LIMIT` time units, and UPPAAL returns the trace reaching that state, in case the query is satisfied.

Note that satisfaction of Queries (10.7-10.9) is guaranteed by the construction of movement and task execution UTA. The queries can still be verified for a given mission, but they are not used in the mission planning. If Query (10.10) is satisfied, UPPAALoutputs the trace reaching the goal state. The trace is processed by TAMAAand a schedule is generated following the steps of the trace. We always look for the fastest trace. When the task execution time is a time interval rather than a fixed value, we assume the environment to be under the control of the agents, which means that the agents can choose any task execution time during the time intervals, respectively. If the environment reacts competitively or possibly antagonistically, we need a comprehensive schedule that considers all possible scenarios. We refer the interested readers to our previous work [16], in which we propose a method combining model checking and reinforcement learning to deal with uncooperative environments.

### 10.4.3 Mission Planning with DALI and TAMAA

Path planning and task scheduling aspects of the mission planning depend on each other. One of the task scheduling parameters is traveling time between pairs of milestones. Temporary obstacles affect shortest paths between milestones, therefore path planning requires to know the starting time points of travels in order to ensure avoidance of temporary obstacles. Therefore, mission planning might re-run the path-planning and task-scheduling modules multiple times until all requirements for the mission are satisfied: changing paths affect the traveling time and, consequently, might affect the task schedule; changes in the task schedule modify the starting time points of travels and, consequently, might affect the paths. In addition, task scheduler might find impossible to satisfy timing constraints with the given traveling time. In this case, DALI can attempt to improve the traveling time by ignoring soft constraints and calling TAMAA to reschedule tasks with the new traveling time as the input.

We illustrate the steps of the Algorithm 5 with the running example depicted in Figure 10.4a. During the first step (line 5), DALI discretizes the entire environment into a Cartesian grid shown in Figure 10.4b and builds a graph of the environment which is used in all consequent path-planning calls. DALI computes the path between every pair of milestones while ignoring temporary obstacles during the first computation (lines 7-9). Indeed, at this point neither the order nor the starting time points of travels are known and the shortest paths between every two milestones are returned. In Figure 10.4b, the red path between milestones $A$ and $B$ would be selected as the shortest even if it passes through a temporary obstacle (orange cells).

At the next step, the model generation module of TAMAA automatically generates the network of UTA that models the agents' movement and task execution (lines 10-13), after which a function named `Scheduling` is invoked to schedule the tasks taking into account the traveling time of the paths (line 14). In the `Scheduling` function, UPPAAL checks the existence of a model execution trace satisfying the Query (10.10) (line 32). Note that queries that formalize other requirements, e.g., Queries (10.7) - (10.9), can also be checked, which ensure the satisfaction of other requirements but do not contribute to the mission plan synthesis. Next, if the query is satisfied and a trace is obtained, we convert the trace into a schedule ordering the movement and task execution actions (line 34). In the `traceParser` function, a schedule that orders the actions of movement and task execution for all the agents are generated and returned to the main function of mission planning (line 14). Note that the returned result of schedule can be empty (i.e., `null`), which indicates the non-existence of mission plans. Since the model checker exhaustively explores the entire state space of the model, the non-existence of mission plans is guaranteed. In such case, the mission planning tries to recompute the paths and the schedule ignoring soft constraints (line 19). If the Query (10.10) cannot be satisfied even without soft constraints (line 21), the algorithm terminates and suggests the users to modify their environment configuration or requirements.

If the `Scheduling` function returns a non-empty result, DALI checks whether the plan happens to cross any temporary anomalies and at which time (line 22). The function *CheckTemporaryObstacles* looks at all the paths involved in the schedule, and, with the knowledge of the starting time of travels, checks that no temporary obstacle is entered during its inaccessibility period. If at least one path crosses a temporary obstacle during its existence, recomputation is run (line 25). Given scheduled starting time of actions, DALI updates paths that used to enter the temporary obstacles. New paths might become longer than the original paths, requiring TAMAA to reschedule tasks. For example, in Fig. 10.4b, if a temporary obstacle is enabled then a different path between $A$ and $B$ is selected (dashed black lines). The new path is longer and might affect the satisfaction of timing constraints. In this case, another path is computed ignoring the soft constraint (solid line). TAMAA is called after each path modification and regenerates a new task schedule (line 14).

**Algorithm 5:** Mission Planning Algorithm

```
1  Function MissionPlanning(Environment env, TS tasks[], Agent agents[],
     Query query)
2      $NUTA\ model$                                  // A network of UTA
3      $Bool\ noTemp := true$
4      $Bool\ softExist = true$
5      $Node\ area[] := CreateGrid(env)$
6      $double\ startTimes[][] := 0$
7      foreach $agent : agents$ do
8          foreach $m1, m2 : milestones$ do
9              $agent.paths[m1][m2] :=$
                 $FindPath(area, m1, m2, agent, startTimes[m1][m2], noTemp, softExist)$
10     foreach $agent : agents$ do
11         $UTA\ movement :=$
             $CreateMovementUTA(agent.paths, agent.speed)$
12         $UTA\ taskExe := CreateTaskUTA(agent.ID, tasks)$
13         $model := compose(movement, taskExe)$
14     $Schedule\ schedule := Scheduling(model, query)$
15     if $schedule == null$ then
16         if $softExist$ then
17             $softExist := false$
18             $double\ startTimes[][] := 0$
19             $goto\ line\ 7$
20         else
21             return $false$                         // No mission plan found
22     if $!CheckTemporaryObstacles(schedule)$ then
23         $noTemp := false$
24         $updateTimes(startTimes, schedule)$
25         $goto\ line\ 7$           // Recompute affected paths with DALı
26     foreach $agent : agents$ do
27         $agent.schedule := distribute(schedule)$
28     return $true$
29
30 Function Scheduling(NUTA model, Query query)
31     $Schedule\ schedule := null$          // Stores the order of actions
32     $Trace\ trace = check(model, query)$
33     if $trace \neq null$ then
34         $schedule = traceParser(trace)$
35     return $schedule$
```

Next, the entire plan of actions are dissolved into several individual mission plans for the agents and distributed to each of them (line 27).

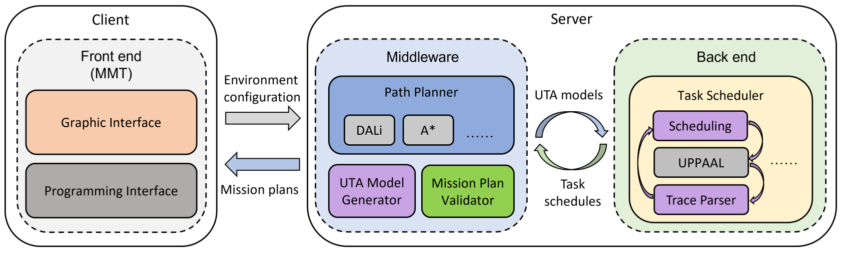


Figure 10.7: Architecture and the information flow of the toolset

The function `distribute` dissolves the entire schedule and puts the actions into the corresponding individual mission plan according to which agent they belong to. Finally, a mission plan that satisfies all constraints imposed on the tasks and paths is returned by the mission planner that integrates DALi and TAMAA. Again, since the satisfaction of constraints is guaranteed by the model checking technique used in the `Scheduling` function, the resulting mission plan is *correct-by-construction*.

## 10.5 Description of the Tool

In this section, we present our toolset called *MALTA*[1] that consists of three main components: a GUI called Mission Management Tool (MMT), a path planner implementing DALi, and a task scheduler implementing TAMAA.

### 10.5.1 Overall Description

The toolset is built of 3 parts: a front end providing the graphic user interface (GUI), a middleware providing path planning and building mission plans from paths and schedules, and a back end dedicated to task scheduling. The toolset design adopts a Client/Server architecture. The reason is twofold: first, the front end of the toolset is a GUI that has been independently designed. Beside the GUI, the front end also provides a group of programming interfaces and data structures for extension and communication. Therefore, the front end is open for extension without

[1] *MALTA* installation package and source code of path planner and task scheduler can be found at https://github.com/rgu01/MALTA

touching its code. Second, the computation of mission plans can be quite expensive. As we show in Section 10.6, synthesizing mission plans for multiple agents can cost hours on a computationally powerful server. Therefore, the separation of front end GUI from the mission plan synthesis allows the users to move computations to a dedicated server, which is a user-friendly and efficient design pattern, also easy to maintain.

In the front end, users can configure their environment including the navigation areas, milestones, tasks, agents, etc., after which, the environmental configuration is transferred to the middleware. The *Mission Management Tool (MMT)* described in the following subsections provides the front end GUI.

The middleware receives the environmental configuration and passes it to a path planner. Any path planning algorithm that supports the desired environmental constraints can be used. In our implementation we offer a choice between A* and DALi algorithms described in Subsection 10.4.1. At the second step the middleware generates UTAmodels following the Algorithms 3 and 4. The *UTA model generator* is based on an open source library *j2uppaal*[2].

These UTA models are transferred to the back end, where we implement the TAMAA scheduler to synthesize schedules. The `Scheduling` module implementing the `Scheduling` function in Algorithm 5 invokes the model checker UPPAAL to check the UTA models against Query (10.10) and, in case of successful verification, UPPAAL generates an execution trace containing the sequence of actions that can be translated into a schedule by the *trace parser*, which uses the library for parsing traces provided by UPPAAL[3]. The back end can use other model-checking-based schedulers, for example missions in uncertain environment could use another scheduler combining model checking and reinforcement learning [17, 16].

The resulting schedule is stored as a standard format of Extensible Markup Language (XML) and sent to the middleware, where the schedule is combined with the paths to generate a mission plan. A module called *Mission Plan Validator* is designed to check if the mission plan happens to come across the temporary obstacles when they still exist. If the collision does happen, a new path plan that circumvents the collision is calculated by the *Path Planner* and the corresponding UTA models that reflect the new paths are generated and sent to the task scheduler

[2]https://github.com/predragf/org.fmaes.j2uppaal
[3]https://github.com/UPPAALModelChecker/utap/wiki

again. The iteration of computation continues until a valid mission plan is generated or no valid path exists. The final mission plans are shown in the frond end when the *Mission Plan Validator* confirms that the results are correct, or a warning informs the users that no mission plan can be generated and suggests a configuration modification.

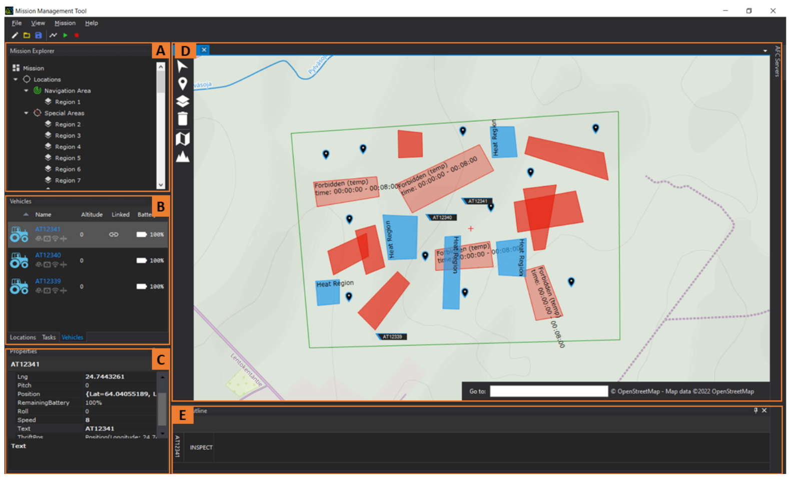
In the following subsections we introduce MALTA's GUI and demonstrate how can one configure the environment, and visualize the resulting mission plan.

### 10.5.2 Mission Management Tool

The Mission Management Tool (MMT) is a GUI that allows the operator to plan, execute and supervise missions involving multiple autonomous vehicles [10]. In the context of this paper the focus is mission definition and plan visualization, hence we do not discuss the plan execution and supervision functionalities.

MMT's main window contains five main panels: (A) mission explorer, (B) assets, (C) properties, (D) map, and (E) plan outline (Fig. 10.8). The map shows an overall view of the mission area and the vehicles. It also provides tools to the operator for defining areas of interest for a mission. The mission explorer (Fig. 10.8.A) represents different assets involved in a mission and their relationship in a tree structure. The assets panel contains three sub-panels that contain vehicles, locations and tasks. These are either physical assets that the operator has access to (vehicles), or entities that the operator has defined himself/herself (tasks and locations). The properties panel shows different properties of a selected asset and allows the operator to set their values if necessary. Finally the plan outline provides a Gantt chart representation of the whole plan for a mission (Fig. 10.9). MMT communicates with the mission planner through the Apache Thrift Framework[4]. This allows MMT and the planner to share definitions of different assets and communicate mission data as well as the final mission plan. Since the version of Apache Thrift that we use (0.9.3) does not fully support asynchronous communication for all the programming languages involved in this work, both MMT and the Planner provide one-way Thrift services to each other for communication. This means that upon a function call through Thrift framework, the client does not wait for a response and instead it provides it own Thrift server for receiving the results (the plan)

[4]https://thrift.apache.org/



AT12341

| | |
|---|---|
| Lng | 24.7443261 |
| Pitch | 0 |
| Position | {Lat=64.04055189, L |
| RemainingBattery | 100% |
| Roll | 0 |
| Speed | 8 |
| Text | AT12341 |

Figure 10.8: Mission Management Tool. (A) Mission Explorer, (B) Assets, (C) Properties, (D) Map and (E) Plan Outline.

whenever they are ready. Hence, the planners run a Thrift server that exposes functions allowing the MMT to send a mission definition to the planner and request a mission plan for it. MMT on the other hand, runs a Thrift server that includes a function, allowing the planner to send the final plan back to MMT.

### 10.5.3 Environmental Configuration with MMT

Mission definition through MMT is done in two steps: First the mission assets and their properties need to be defined, and next the relationship between these assets should be configured. Mission assets consist of vehicles, tasks and locations. Vehicles are not defined by the operator as they are real entities that are discovered by MMT. In case of vehicles, the operator can set some of their properties that might affect the visualization of vehicle on MMT (i.e. color) or a vehicle's performance in a mission (i.e. speed).

The two other types of assets need to be defined by the users. Defining locations/areas of interest for a mission takes place through the map. The operator can use map tools to set markers on the map or draw regions on it. The locations/areas are added to the locations sub-panel, where they can be accessed to set their visualization or mission-related properties. For an area, there are some important properties that can be set, which affect the planning phase:

- *Region Type*: This can be set to a forbidden area (vehicles should not go through that region), a preferred area (vehicles are preferred to go through that region), a less preferred area (vehicles can go through there but it is better to avoid it) and a heat area (hard to pass area).
- *Intensity*: This parameter indicates the intensity of heat areas and (less-)preferred areas. For heat areas it affects the speed drop in the area; for (less-)preferred areas it affects the decision for entering the areas.
- *Start and End Time*: These parameters define the time interval relative to the mission start when the area is active, e.g., temporary obstacles appear and last a while.

Defining the tasks of a mission is done through the tasks sub-panel. This panel allows the operator to define a new task or re-use a predefined one. Defining new tasks requires the operator to define the task type (either *inspect* or *survey*) and the equipment required for it. Inspect tasks are tasks that should be performed on a location (i.e. digging), whereas, survey tasks are performed on a whole area (i.e. spraying a field).

When all the assets are defined, the operator can define the mission by dragging and dropping the assets to the mission explorer. The mission explorer contains several entry points for mission assets as follows:

- *Navigation Area*: This is the main area of the mission. No vehicle will be allowed to move outside this area. Navigation area is visualized as a polygon with green borders in MMT (Fig. 10.8.D).
- *Special Areas*: These are the areas that have specific roles in the mission but are not part of a task. Examples include: forbidden areas, preferred areas, and heat areas. In MMT, forbidden areas are displayed with a red background and if they are temporarily forbidden a lighter shade of red is used. Temporarily forbidden areas are also annotated with a timespan during which they become inaccessible by the vehicles. Heat and preferred areas are displayed in light blue (Fig. 10.8.D).

- *Task Areas*: These are areas that are related to a task. For each task involved in the mission a new entry point is added, allowing the operator to add the locations/areas related to it. It is possible to add several locations/areas to a single task. If a task requires a specific sensor/actuator, this information is also written next to it on the map (Fig. 10.8.D).
- *Home Locations*: These are the locations where the vehicles should move to after finishing their mission.
- *Vehicles*: This part contains all the vehicles that are allowed to participate in the mission. This means that the operator is allowed to only drag some of the vehicles to this section which in turn means the planner can only use those for planning (Fig. 10.8.B).
- *Tasks*: This contains a list of all the tasks that should be performed in the mission. When the operator drags a task to this section, a new sub-section for this task is also added to the *Task Areas* section, allowing the operator to define the areas for this specific task (Fig. 10.8.B).

The final step in mission definition is defining location/area properties after they are added to a task. Please note that these properties are not directly bound to the location or the task itself, but are properties that represent that specific task while being done at that specific location. These properties are only accessible by clicking on the location/area under the task in task areas section of mission explorer (Fig. 10.8.C). These properties allow the operator to define task preconditions, BCET and WCET.

After defining the mission, the operator can save it for later use or send it to the planner using the plan button on the toolbar. This sends the plan to the planner and awaits for a result. The final mission plan is then visualized on the map and plan outline.

### 10.5.4 Mission Plan Demonstration in MMT

Fig. 10.9 shows a plan visualized in MMT. The plan contains only one vehicle and nine tasks. The vehicle and its related path are color-coded (in this case they are all green). In cases with multiple vehicles, each vehicle, its related path and tasks will get a separate color. It can be observed that red areas are always avoided by the green lines (vehicle's path), while the light-red areas are sometimes avoided (during the related time span) and sometimes crossed (when the area is not forbidden anymore). The blue regions indicate areas with bad roads: vehicles can pass them but at a slower speed. The plan outline at the bottom also shows the order of actions to be taken by the vehicle and the amount of time each action consumes.

The mission starts from the initial location of the vehicle and the path shows the milestones that the vehicle will visit during its mission. Some milestones are visited several times as this is part of the task and mission definition. The milestone locations are marked with markers matching the color of the vehicle and a text shows the actual action which will be performed at that location.

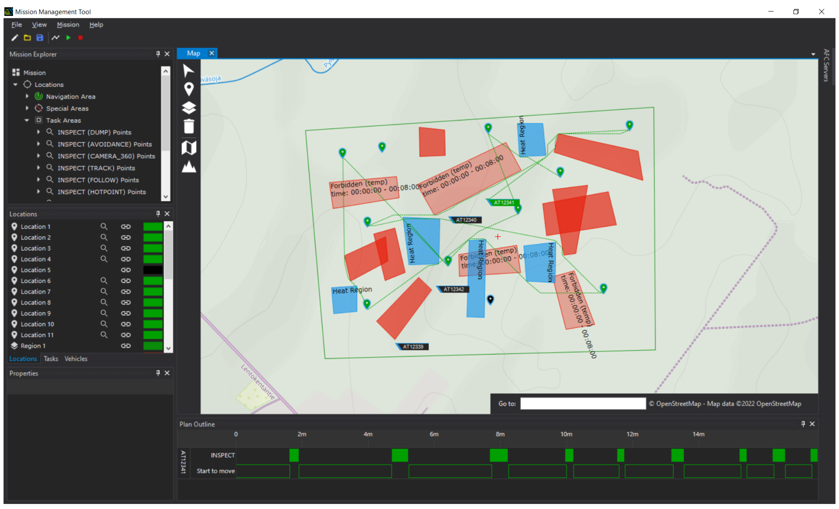


Figure 10.9: Mission Management Tool.

## 10.6 Evaluation

To test our prototype implementation we conducted a series of experiments directed to evaluate the performance and scalability of the proposed approach[5]. In this section, we present the design and results of the experiments.

### 10.6.1 Methodology

For the experiments we have created a mission shown in Figure 10.8 involving multiple milestones, permanent and temporary obstacles, and heat areas. The following parameters have been varied in the experiments.

- *Path-planning algorithm*: we compared A* algorithm, DALI without optimizations, and DALI* discussed in Section 10.4. We refer to the version without optimizations as DALI. All three algorithms have similar implementations being different only in the distance computations for the node selections, respectively, thus the comparison has no bias in implementation.

- *Granularity of a partition of the navigation area into nodes*: during the transformation of the navigation area into a graph, the parameter (namely, granularity henceforth) sets the sizes of Cartesian grid cells and the distance between neighbour nodes. A smaller distance results in finer granularity and a larger graph. We use the granularity in range $[2, 10]$ resulting in approximately 15000 nodes in the graph for the granularity 10 and 390000 for the granularity 2.

- *Presence of temporary obstacles*: in a part of the experiments where the performance of DALI or DALI* is compared with A*, we ignore all temporary obstacles.

- *Number of tasks/milestones* is in range $[1, 10]$.

- *Number of permanent obstacles* is in range $[1, 10]$.

- *Number of heat areas* is in range $[1, 5]$.

- *Number of vehicles* is in range $[1, 4]$.

[5]The mission configurations of the experiments are published so that one can replicate the experimental results: https://github.com/rgu01/MALTA.

In order to vary the numbers of milestones, permanent obstacles, and heat areas and to avoid the generation of a separate mission for each combination of the parameters, we use the following strategy. The mission contains a set of milestones (permanent obstacles, heat areas) and in each experiment we select a subset of all milestones (permanent obstacles, heat areas) of a desired size (ensuring that all task preconditions can be met) and perform mission planning with the selected milestones (permanent obstacles, heat areas).

In all experiments we compute the time needed to generate a graph, the total time used by the path-planning algorithms, and the time used by TAMAA (i.e., calls to UPPAAL). Each experiment involves multiple calls to the path-planning algorithm and to UPPAAL; we compute the total time for all calls. All experiments have been conducted 5 times and we consider the mean time as a result. The front end and the middleware are on a same PC with a 12-core i7 CPU, 16 GB RAM, and Windows 10 OS. The back end is on a server with a 48-core CPU (Intel Xeon E5-2678), 256 GB RAM, and Ubuntu 18.04 OS. The timeout of computation is set to be 1 hour in the experiments.

We perform a preliminary experiment comparing two optimizations of DALi. For the preliminary experiment, we use a single vehicle and remove temporary obstacles and heat areas from the mission to ensure that path planning is called only once. We vary the granularity and the number of milestones, and compare the time taken by the path-planning algorithms. The experimental results show that on our mission DALi* is faster than the single-source multiple-target optimization. Therefore, in the remaining experiments we do not use the single-source multiple-target optimization of DALi. The results of this preliminary experiment are discussed in Subsection 10.6.2.

The experiments have been divided into 4 groups shown as follows:

1. *Group I*: agent amount: 1, presence of temporary obstacles and heat areas: false, path-planning algorithms: A*, DALi and DALi*.

2. *Group II*: agent amount: 1, presence of temporary obstacles and heat areas: true, path-planning algorithms: DALi and DALi*.

3. *Group III*: agent amount: $1-4$, presence of temporary obstacles and heat areas: false, path-planning algorithms: A*, DALi and DALi*.

4. *Group IV*: agent amount: $1-4$, presence of temporary obstacles and heat areas: true, path-planning algorithm: DALI*.

Group I for evaluating the performance difference between A* and DALI algorithms (i.e., DALI and DALI*). Group II considers the effect of heat areas and temporary obstacles on the performance of DALI algorithms. The A* algorithm is not used in this group of experiments since it does not support navigation in the presence of heat areas and temporary obstacles. Groups III and IV consider multiple vehicles to see the influence of the vehicle numbers on the performance of the DALI algorithms.

In the Figures 10.10-10.21, we show the influence of 1 or 2 parameters on the execution time of our tool, while the remaining parameters are set to default values: 1 vehicle, 10 milestones, 10 obstacles, 0 heat areas, and granularity 4.

### 10.6.2 Comparison of DALI optimizations

In this preliminary experiment, we compare the path-planning time of the basic DALI algorithm and two optimizations proposed in Subsection 10.4.1. The first optimization converts DALI into the single-source multiple-target algorithm. Considering that our methodology requires to compute paths between every pair of milestones, such optimization drastically reduces the number of calls to the algorithm. The second optimization referenced as DALI* uses an heuristic from the A* algorithm.

The results show that both optimizations are significantly faster than the original DALI algorithm. Among the two optimizations, the DALI* is clearly the fastest. The results for the granularity set to 4 are shown in Figure 10.10. Considering the results of this experiment, we do not use the single-source multiple-target optimization in the following experiments.

### 10.6.3 Comparison of A* and DALI

In the first group of experiments we create a mission plan for a single vehicle with the A* algorithm and 2 versions of the DALI algorithm without a consideration of heat areas and temporary obstacles. We vary the number of milestones, the number of permanent obstacles, and the granularity. We report the mean value of the execution time of 5 runs. The time spent on the graph generation (Figure 10.11) and

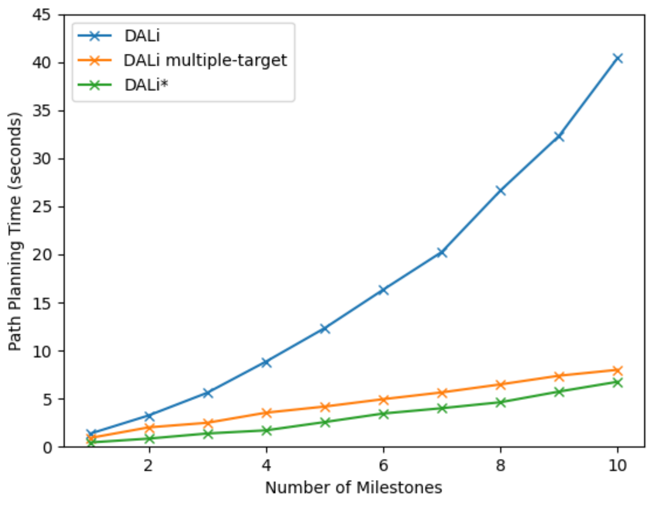


Figure 10.10: Comparison of DALi and its optimizations

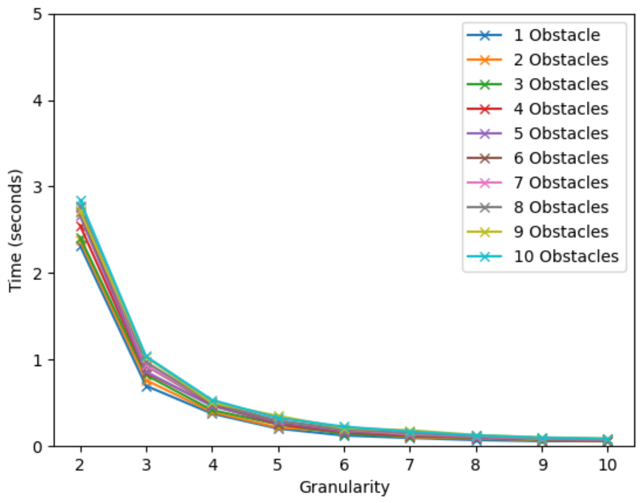


Figure 10.11: The first group of experiments: the graph generation time w.r.t. the granularity and the number of obstacles

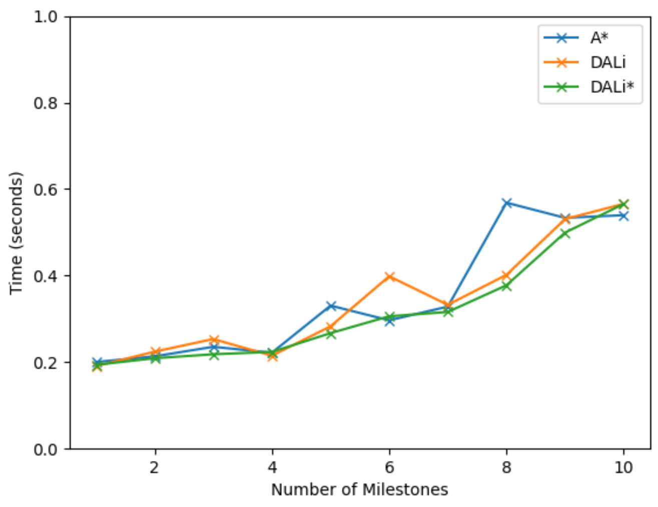


Figure 10.12: The first group of experiments: the UPPAAL call time w.r.t. the number of milestones

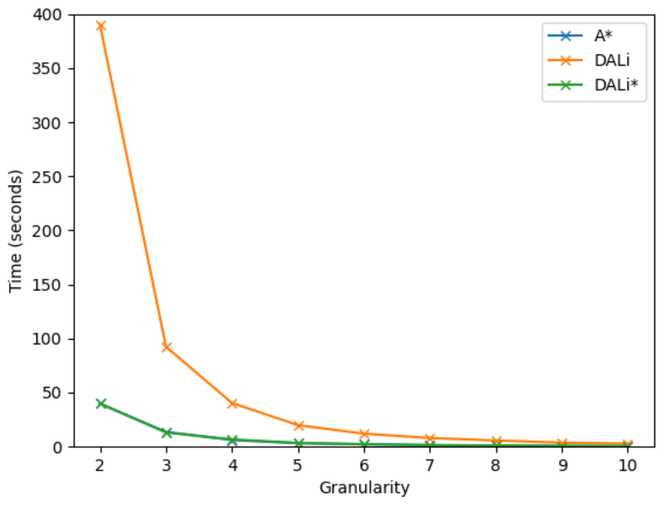


Figure 10.13: The first group of experiments: the path-planning time w.r.t. the granularity

on the UPPAAL call (Figure 10.12) is independent from the choice of algorithms. The graph generation time depends on the granularity since the number of nodes in a graph is inversely proportional to the square of the granularity. In this group of experiments, only a single call to UPPAAL is performed. The lack of temporary obstacles implies that the generated plan can satisfy all constraints after the single UPPAAL call. The number of milestones affects the size of the UPPAAL model and, consequently, the time needed to synthesize a plan. The computation time of all path-planning algorithms depends on the graph size, therefore the finer granularity the more time is used to compute paths (Figure 10.13). Since paths have to be computed between each pair of milestones, the total path-planning time grows with the number of milestones (Figure 10.14). On both figures we can notice that A* and DALI* have the same performance; whereas the original DALI is significantly slower and is more drastically affected by the granularity and the number of milestones. It is interesting to note that while A* and DALI* are barely affected by the number of obstacles, DALI performs 20% faster with ten obstacles than that with a single one (Figure 10.15). Indeed, obstacles reduce the number of nodes in the graph, which positively affects the path search time. Computation time of DALI* and A* are less affected by this parameter: first of all, they scale better with the number of nodes than DALI does and, therefore, a small reduction in the number of nodes has a minor effect on the algorithms' performance. Moreover, the heuristic used in A* and DALI* uses the distance estimation to the target node and obstacles on the path make the estimation less reliable.

### 10.6.4 Evaluation of the Approach with Heat Areas and Temporary Obstacles

In the second group of experiments, only two versions of DALI are evaluated, because the A* algorithm cannot take heat areas and temporary obstacles into consideration. The mission has been specifically designed to ensure that paths computed before the first call to UPPAAL would cross the temporary obstacles. Therefore, the generation of a correct mission plan requires a recomputation of several paths and additional calls to UPPAAL.

The graph generation is unaffected by the presence of temporary obstacles as they are not considered during the graph construction, however

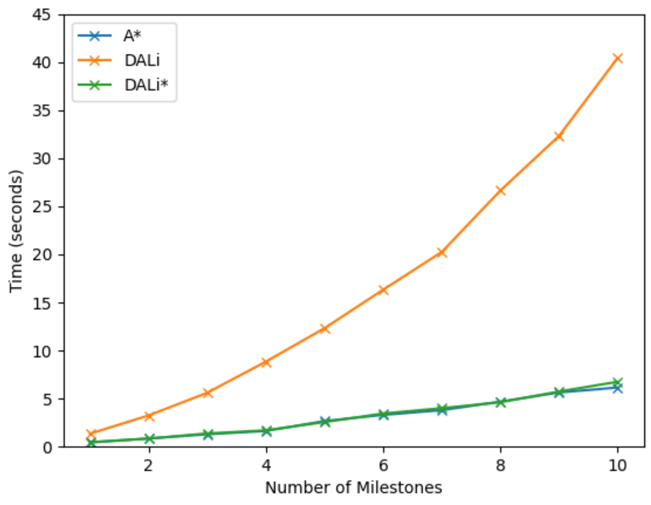


Figure 10.14: The first group of experiments: the path-planning time w.r.t. the number of milestones

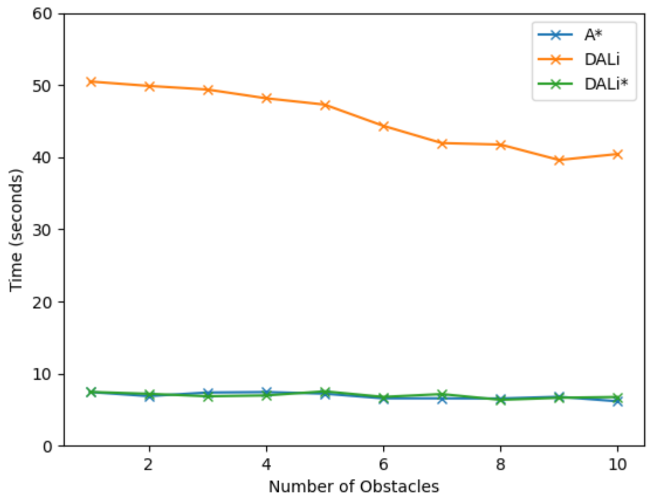


Figure 10.15: The first group of experiments: the path-planning time w.r.t. the number of obstacles

nodes located inside the heat areas have to be marked. Thus, in Figure 10.16 we can notice that the graph generation time slightly rises with the number of heat areas. The time for a single UPPAAL call has not been affected by the presence of heat areas and temporary obstacles. Figure 10.17 shows the results for the two versions of DALi. For comparison, we add the corresponding results from the previous group of experiments without temporary obstacles. Due to the presence of temporary obstacles, TAMAA requires up to 4 additional calls to the path-planning algorithm and up to 2 additional calls to UPPAAL, though its effect on the computation time of path planning is minor: a few seconds for DALi and less than 1 second for DALi*. Indeed, for 1 vehicle all UPPAAL calls require less than 0.6 seconds and the path-planning calls (at granularity 4) take about 0.5 second for DALi and 0.07 seconds for DALi*, respectively (Figure 10.18). The presence of heat areas does not have a significant effect on the path-planning time (Figure 10.19). Nevertheless, heat areas can change the shortest path and, as a result, affect the number of recomputations of paths: a new path can lead towards a temporary obstacle or, conversely, can help to avoid it. In the experiment, we have the latter case: one of the heat areas changes a shortest path between the start point and the first milestone and avoids a temporary obstacle. As a result, the recomputation is not called, thus the path-planning and total time are smaller than those of the initial computation. In Figure 10.19, the path planning for 1 milestone and 5 heat areas takes only 0.4 seconds in comparison to 0.7 seconds for 1 milestone and $0 - 4$ heat areas. Note that the path-planning for 4 or more milestones and 5 heat areas is not significantly faster than for $0 - 4$ heat areas: paths between other milestones cross the temporary obstacles causing the recomputation.

### 10.6.5 Results for Multiple Agents

In the third group of experiments, we evaluate how the tool performs with multiple agents. Within this group, the mission with temporary obstacles is used and the only considered path-planning algorithm is DALi*. Agents have different speeds and subsets of tasks to do. We assume that all agents can drive through any non-obstacle area and the heat maps have the same slowing factor for all agents. Therefore, without the consideration of temporary obstacles, the shortest path between a pair of milestones can be the same for all agents.

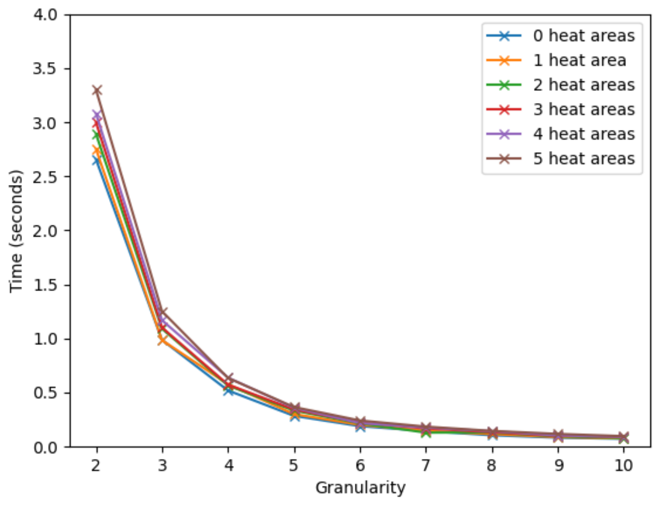


Figure 10.16: The second group of experiments: the graph generation time w.r.t. the number of heat areas

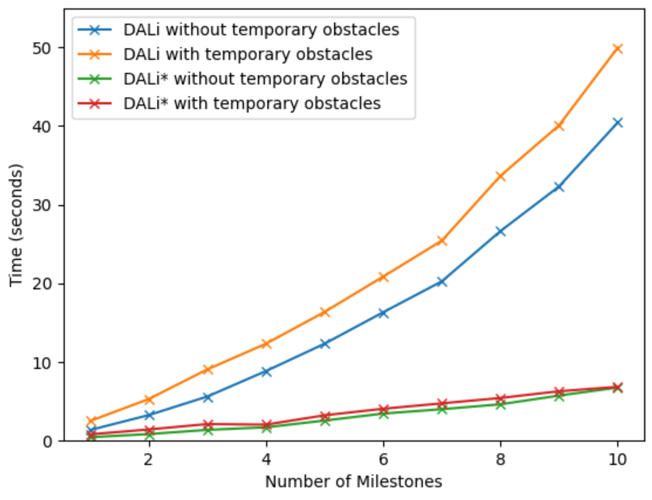


Figure 10.17: The second group of experiments: the path-planning time w.r.t. the number of milestones

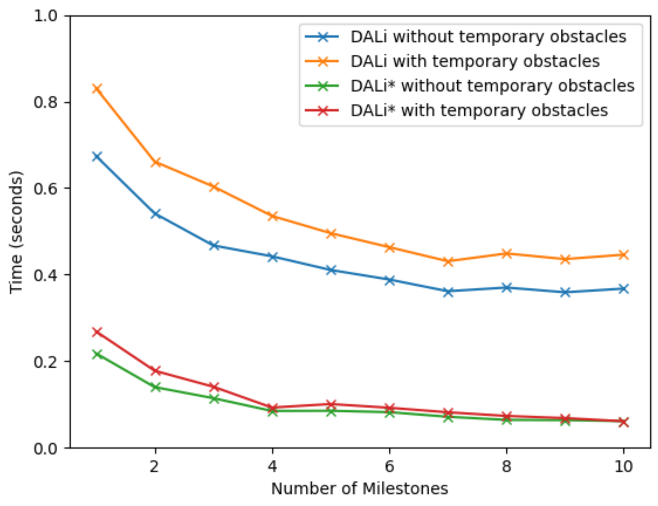


Figure 10.18: The second group of experiments: the mean path-planning time

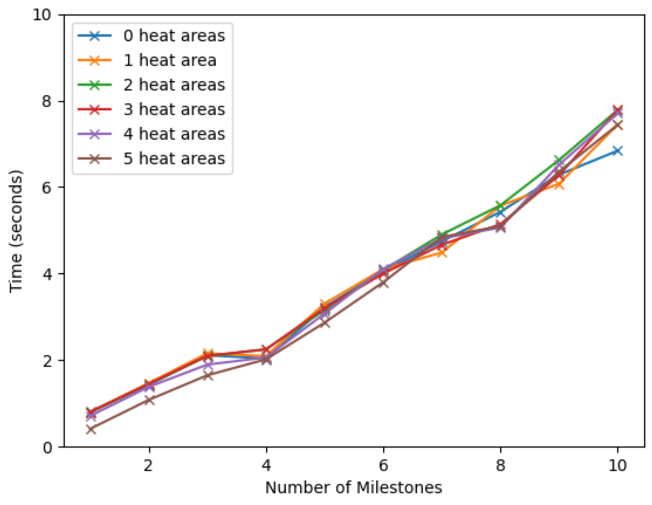


Figure 10.19: The second group of experiments: the path-planning time w.r.t. the number of heat areas

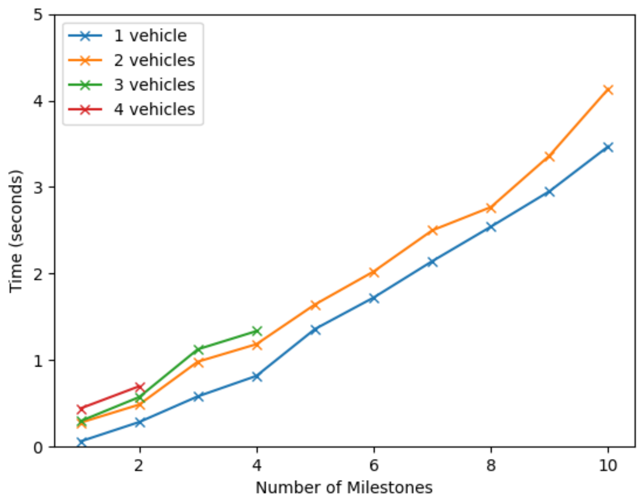


Figure 10.20: The third group of experiments: the path-planning time w.r.t. the number of milestones

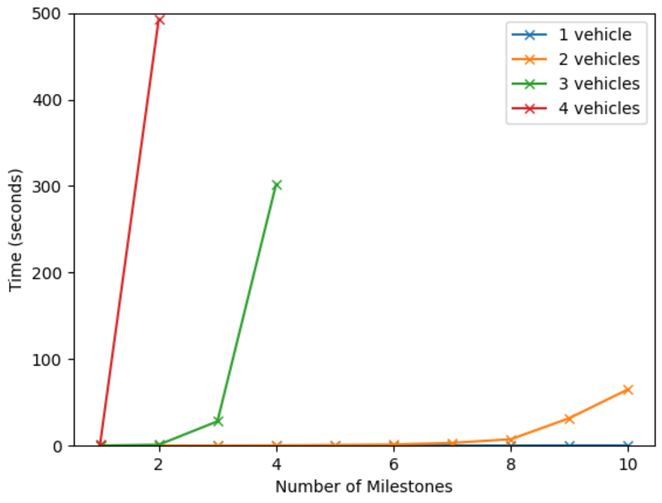


Figure 10.21: The third group of experiments: the UPPAAL time w.r.t. the number of milestones

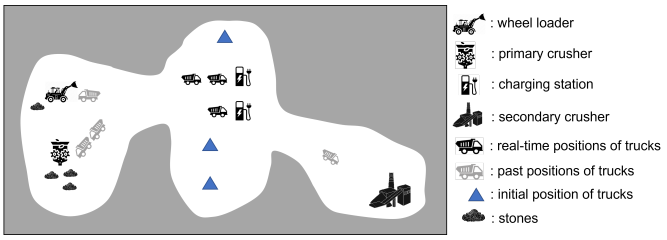

: wheel loader
: primary crusher
: charging station
: secondary crusher
: real-time positions of trucks
: past positions of trucks
: initial position of trucks
: stones

Figure 10.22: An example of the autonomous quarry

An involvement of multiple agents in the mission has low effect on the path-planning algorithm performance. The assumption above on the common shortest paths for agents allows us to reuse paths between milestones for all agents, thus an addition of an extra agent only requires to compute paths from its starting location to other milestones. The number of UTA used in TAMAA and, consequently, the complexity of the composed model of multiple agents depend on the number of agents.

Figures 10.20 and 10.21 show the time required by DALI* and by UPPAAL, respectively. Our tool times out during UPPAAL calls with no result in cases of 3 agents with 5 milestones, and 4 agents with 3 milestones. Therefore, figures 10.20 and 10.21 do not show the computation time of these cases.

Introduction of temporary obstacles that requires additional calls to UPPAAL roughly multiplies the computation time by the number of calls to UPPAAL. Our previous work [17] proposes a method named MCRL (model checking + reinforcement learning) to solve this scalability problem caused by large numbers of agents. We leave the integration of MCRL into MALTA as a future work.

## 10.7 Adaptability of MALTA: a Special Industrial Use Case

Variability of mission planning problems is immense. Even within the autonomous quarry case study, there is a huge spectrum of requirements starting from safety properties to liveness properties [18], such as agents must never go across a certain area when humans are working there (safety property), and agents should repetitively enter the charging points until they accomplish the mission (liveness property). Due to high variability, it is infeasible to build a fully automated solution that fits all possible cases efficiently. Therefore, the adaptability of a solution plays a crucial role, which requires an easy way of adapting the agent models and queries to different applications and their requirements.

In this section, we show an example of a variant of our industrial case study: the autonomous quarry, which is sightly different from the problem definition (i.e., Definition 7). We explain how to adapt MALTA to the special use case depicted in Fig. 10.22. The quarry has 3 identical autonomous trucks transferring stones from a primary crusher to a secondary crusher. Stones are gathered at the left side of the quarry and can be loaded into the trucks either by the primary crusher or by a wheel loader. The primary crusher is required to minimize idle time, therefore the wheel loader can only be used if the primary crusher is already occupied by two trucks: one is being loaded and another is waiting in a queue. Stones should be unloaded from the trucks at the secondary crusher on the right part of the map. On the way between crushers there are two charging stations. The use case requires trucks to stop for charging every time they pass the charging stations, no matter how much battery they have left. Each charging stop takes 30 seconds.

The use case has 3 trucks, 1 wheel loader, 1 primary crusher, and 1 secondary crusher. Besides the special charging task, there are 2 regular tasks for trucks: loading stones that can either be performed at the primary crusher or at the wheel loader, and unloading stones at the secondary crusher.

Table 10.2: Machine parameters in the autonomous quarry

| | Machine | Speed | Rate | Capacity |
|---|---|---|---|---|
| Mobile | Autonomous truck | 35 km/h | 1.5 tons/s | 15 tons |
| Stationary | Wheel loader | / | 1.5 tons/s | / |
| | Primary crusher | / | 0.25 tons/s | / |
| | Charging station | / | 30 s/time | / |

Table 10.2 shows the parameters of the machines in the quarry. Trucks can carry 15 tones of stones, and the primary crusher can load 0.25 tons of stones per second, therefore the primary crusher takes 60 seconds to fill in one truck, while the wheel loader can do that in 10

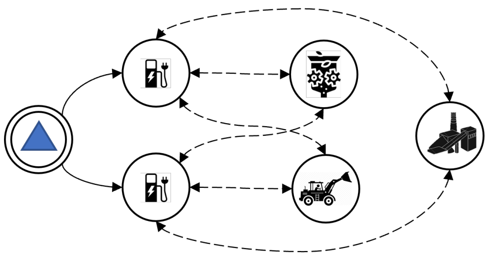
seconds. The trucks can unload 1.5 tons of stones per second, so trucks unloading a full bucket of stones into the secondary crusher takes 10 seconds. The mission goal is to transfer 90 tons of stones as fast as possible.

We use MALTA to create a mission plan for the trucks. The use case does not specify distances in the quarry, therefore we assign them in a manner that ensures that the trucks can finish the tasks without draining out. For path planning, the DALI* algorithm is applied. However, the use case imposes constraints on tasks that have not been covered in Section 10.4, in particular the special task of charging and priority between primary crusher and wheel loader. Therefore, we need to adjust the already introduced UTA models and queries for this use case.

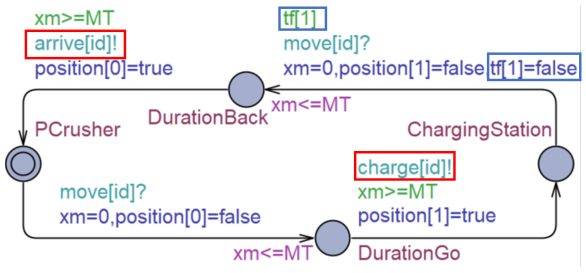
## 10.7.1 Adjustments of the Models

The use case has two requirements that are not supported directly by the original model generated by MALTA. Besides, the topology of the map is also changed to match the special geographic arrangement of machines in the quarry. In this section we explain the modifications necessary to incorporate the new requirements.

The original movement UTA assumes a direct connection between every pair of milestones. However, the geographic arrangement of the use case is different. As the charging stations occupy the center of the quarry, trucks must pass them when traveling from the left to the right of the quarry, and vice versa. Hence, the topology of the quarry for this use case is adjusted. As depicted in Fig. 10.23a, the primary crusher, wheel loader, and the secondary crusher are connected via the charging stations. The generated movement UTA enforces the topology, and thus only accepting the traveling between the charging stations and other milestones. From a truck's point of view, the wheel loader and primary crusher are both for loading stones, so there is no need to connect the wheel loader and primary crusher in the topology. The new requirement of the charging task also induces other adjustments of the movement UTA. Fig. 10.23b shows a part of the adjusted movement UTA, where changes are highlighted by the blue and red squares. When a truck, which is identified by variable `id`, leaves a charging station, its charging task, which is represented by `tf[1]`, must be executed. After the truck leaves the charging station, the variable `tf[1]` flips to *false* so that the next time it reaches a charging station, the charging task can be carried



(a) The topology of the autonomous quarry in Fig. 10.22



(b) A part of the adjusted movement TA

Figure 10.23: Adjustments of the model for the industrial use case.

out again.

The second additional requirement of this use case is the priority between the primary crusher and the wheel loader. To fulfill such requirement, an adaptation of the task execution UTA is necessary. As depicted in Fig. 10.24, we add an additional constraint $PCQueue > 2$ to the guard of the *edge* going to *location* `T2_2` (representing task "Loading at the wheel loader), where *PCQueue* is a global variable counting the number of vehicles at the primary crusher. Therefore, the adapted task execution UTA models that when the length of the waiting queue at the primary crusher is less than two, trucks must go to the crusher; otherwise, the trucks can choose to wait in the queue or go to the wheel loader for loading stones. The synthesized mission plan is supposed to make a wise choice in this case for the optimal productivity. In addition, in the movement UTA, *edges* going into and leaving from the location representing the primary crusher updates the *PCQueue* variable.

The mission of this use case is defined to carry all the stones to the

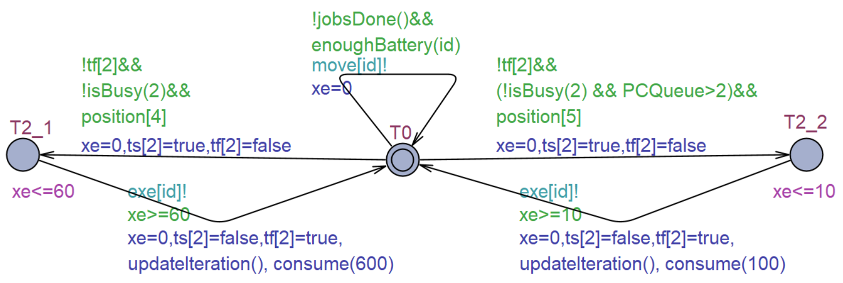

!jobsDone()&&
enoughBattery(id)
move[id]!
xe=0
!tf[2]&&
!isBusy(2)&&
position[4]
xe=0,ts[2]=true,tf[2]=false
T2_1
T0
T2_2
!tf[2]&&
(!isBusy(2) && PCQueue>2)&&
position[5]
xe=0,ts[2]=true,tf[2]=false
xe<=60
exe[id]!
xe>=60
xe=0,ts[2]=false,tf[2]=true,
updateIteration(), consume(600)
exe[id]!
xe>=10
xe=0,ts[2]=false,tf[2]=true,
updateIteration(), consume(100)
xe<=10

Figure 10.24: A part of the adjusted task-execution UTA

secondary crusher rather than complete all the tasks a desired number of times. Therefore, we add the auxiliary variables *stone* and *load*, which represent the total volume of stones remained to be transferred and the vehicles capacity, respectively.

### 10.7.2 Additional Adaptation of Queries and Models

The requirements formalised by Queries (10.7) - (10.9) are not changed for the use case, however the Query (10.10) is replaced due to the different mission formulation by:

$$\texttt{E<> stone == 0} \tag{10.11}$$

The use case has two additional requirements:

1. Prioritizing the primary crusher before the wheel loader: a truck can choose the wheel loader only in case when there are two other vehicles at the primary crusher (one being served and one in a queue).

2. Battery charging: whenever a truck passes by a charging station, it must stop there and charge for 30 seconds. The trucks' batteries must never be consumed before they finish the global mission.

The former requirement has a straightforward encoding into the following UPPAAL query:

$$\texttt{A[] PCQueue < 2 imply !(task}_0\texttt{.T2\_2 || ... || task}_n\texttt{.T2\_2),} \tag{10.12}$$

This query checks that at any moment in case of the queue at the primary crusher being not full, the task execution UTA of any truck is not at the *location* `T2_2` that represents being loaded at the wheel loader.

The latter requirement consists of two parts. First, it requires the trucks to always charge themselves right after they arrive at a charging station. Second, it requires the trucks to charge themselves timely so that their batteries are not consumed. A logic formalization of the first part is called "*always next*", i.e., the *next* action of moving to a charging station is *always* charging. Unfortunately, the "*always next*" property cannot be formalized in the query language supported by UPPAAL (a subset of TCTL [9]). To overcome this difficulty, we design an auxiliary UTA, to help us verify this requirement. Fig. 10.25 shows the auxiliary

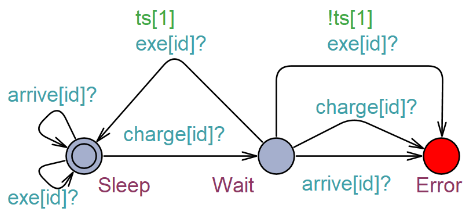


Figure 10.25: The auxiliary UTA *monitor* for verifying the repetitive charging requirement.

UTA, namely *monitor*, where *channels* `arrive` and `charge` synchronize the movement UTA and the monitor UTA (see Fig. 10.23b too). The *channel* `exe` is for synchronizing the task execution UTA and the monitor UTA when the truck is going to execute a task (see Fig. 10.24 too). Therefore, the monitor UTA initially stays at *location* `Sleep` when the truck moves and executes tasks (i.e., two self-loops at *location* `Sleep` in Fig. 10.25). If the truck is moving to a charging station, the monitor UTA transfers to *location* `Wait`, and synchronizes with the movement UTA via *channel* `charge`. Next, the monitor UTA has multiple choices of the next transition: charging or executing other tasks (i.e., synchronization via *channel* `exe`), or moving to other milestones (i.e., synchronization via *channel* `charge` or `arrive`). A Boolean variable `ts[1]` is used to indicate whether the current truck is charging or not. The *edge* from *location* `Wait` to *location* `Sleep` is guarded by this variable, meaning that only charging can make the monitor UTA go back to its initial *location*, whereas other transitions will end up to a *location* representing errors. In summary, the monitor UTA regulates the correct order of a truck's task execution, i.e., moving to a charging station must be always succeeded by charging. If the trucks violate this regulation, they will be stuck at the `Error` *location*. The contraposition of this statement forms the "*always next*" constraint in the battery-charging requirement: if the truck models never visit the `Error` *location*, the next action of moving to a charging station is always charging. Now, we encode an invariance query to verify this property:

$$\texttt{A[] !monitor}_0\texttt{.Error \&\& ... \&\& !monitor}_n\texttt{.Error} \tag{10.13}$$

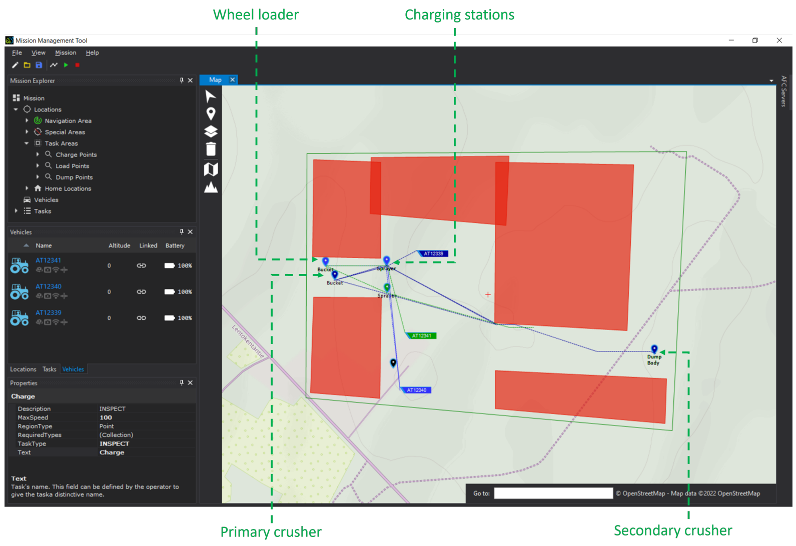
Query (10.13) requires that *monitors of trucks* never reach the `Error` *location*. Satisfying this query indicates that the trucks always charge themselves after arriving at a charging station.

To verify the second part of the battery-charging requirement, we need to add an array of integers to store the battery levels of trucks. When a truck moves, the corresponding integer of its battery level decreases. The consumption rate is assumed to be proportional to the truck's traveling time. If the truck charges, the integer increases. The CTL query that encodes this property is as follows, where `N` is the index of the last truck.

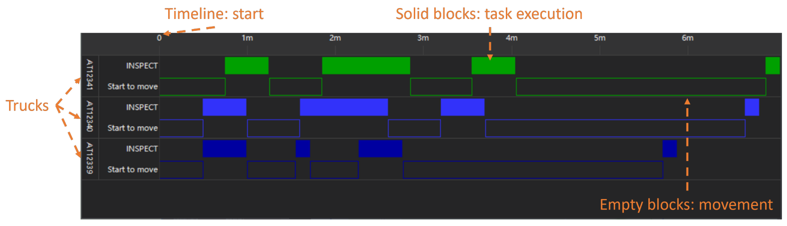
$$\texttt{A[] forall(i:int[0,N]) battery[i]>0} \tag{10.14}$$

### 10.7.3 Synthesis Results

Given the machine parameters shown in Table 10.2, we use *MALTA* to synthesize a mission plan that controls the three trucks to transport 90 tons of stones to the secondary crusher while satisfying all the requirements of the use case, and find the fastest way to finish the global mission. After running *MALTA* to generate the fastest trace that satisfies Query (10.11), a synthesized result of mission plan is illustrated in *MMT* (Fig. 10.26). Fig. 10.26a shows the paths of avoiding static obstacles and visiting different milestones. The task schedule (Fig. 10.26b) shows the interleaving sequence of movement (empty rectangles) and task execution (solid rectangles). One can check the details of path plans by clicking on the empty rectangles, and then the corresponding path plans will be highlighted in the GUI of path plans. The mission plan shows that the fastest time of transporting all the stones is 6.9 minutes. The fact that



(a) The path plans of trucks



(b) A part of the Gantt chat presenting the task schedules of trucks

Figure 10.26: A mission plan of trucks working in the autonomous quarry of Fig. 10.22

a mission plan is depicted in *MMT* demonstrates that Queries (10.12) - (10.14) are satisfied.

## 10.8 Related Work

Path planning, a.k.a., motion planning in the Artificial Intelligent (AI) community, has been an interest of research since the early days of robotics [19]. Sampling-based methods like Rapidly-exploring Random Tree (RRT) [3] and a method based on probabilistic roadmaps for path planning [20], and graph-search-based methods like A* [2] and Theta* [21] are two typical branches of path-planning algorithms. The main contribution of these algorithms is to find collision-free paths in a static and continuous world, in which the topology of the moving space does not change. Moreover, when a robot starts to interact with the world, e.g., picking an object and carrying it to another position, the robot's and object's dynamics and kinematics are changed, which can cause the initial motion plan to be unsuitable. Alami *et al.* [22] and Hauser *et al.* [23] propose a modal structure of the robots and their working environments. Since the switch of modes is discrete, the problem is about identifying the modes of the systems, defining the transitions among the modes, and traversing the state spaces in order to find a trace that satisfy some certain constraints. This is the so-called task planning in the AI community [24]. These approaches do not guarantee correctness unless coupled with a formal verification technique.

Integrating task and motion planning (TAMP) provides us a good understanding of our problem: hybrid discrete-continuous search problem [25]. Research in this area often combines AI and robotics and seeks to provide a separation of concerns by designing hierarchical frameworks, in which high-level task planning and low-level motion planning are separated into different layers, and connected via an intermediate layer [11, 26]. Downward refinement in the methods proposed by Bacchus *et al.* [27] and Nilsson *et al.* [19] first plans at the high level and then refines the high-level plans to low-level ones. The authors assume that their problems fulfil the *downward refinement property* [27], which is often not the case in reality. Our method, on the contrary, starts from the low level to calculate path plans that are collision-free, and then integrates the path-planning results into the model for task planning. Therefore, our task-planning results are naturally collision-free.

There is an important line of work of task planning that uses temporal logic to specify the high-level requirements of tasks [28, 29]. Linear Temporal Logic (LTL) is the most widely used logic for requirement specification [30, 31, 32], because of its expressive power that is able to capture relatively complex requirements, such as repetitively filling the water tank if the water level is lower than a certain level. Different from these studies, we adopt Timed Computation Tree Logic (TCTL). (T)CTL and LTL are members of a temporal logic family named CTL*

[18]. Each of them has its own expressive power and thus is used in different problems. TCTL enables one to express timed requirements such as digging 1000 $m^3$ of stones per 24 hours, which is of high industrial concern, in an attempt to ensure productivity when using autonomous vehicles. Most importantly, our task planning is fully integrated with path planning. Therefore, the results of our method comprehensively consider both the traveling time, as well as the task execution time and order.

In the formal methods community, task planning is being challenged by various formalisms and methods. When the formalisms only have stochastic models, the problems fall into a category called $\frac{1}{2}$-player games [15]. In $\frac{1}{2}$-player games, neither agents nor environments get the control of their behaviors and the corresponding outcome, e.g., flipping a coin. By replacing the stochastic behaviors of agents with non-deterministic choice of actions, $\frac{1}{2}$-player games are changed to 1-player games, which is the problem that we are solving in this paper. Note that, 1-player games assume that the environment is fully controlled by the agents, so that the winning strategies are totally dependent on the behaviors of the agents [15]. Besides UPPAAL, there are many other tools that aim to solve this kind of problems, e.g., Kronos [33], LTSim [34], and SpaceEx [35]. The major difference between our tool and these mentioned ones is that our MALTA tool integrates path-planning algorithms and task-scheduling algorithms, and has a dedicated GUI for mission planning, which provides interfaces for extension. Adding stochastic behaviors to environments makes the formalism represented as a $1\frac{1}{2}$-player game, and changing the stochastic behaviors of environments into non-deterministic ones that are independent from agents makes the formalism to be a 2-player game [15], both of which are out of the scope of this paper. There are studies that investigate synthesizing controllers from various temporal logic specifications. Alur *et al.* [36, 37] propose a compositional method for synthesizing reactive controllers satisfying Linear Temporal Logic specifications for multi-agent systems. Tumova et al. [38, 39] present their method for motion planning of multiple-agent systems using Metric Interval Temporal Logic (MITL). Inspired by these works, our study aims to bring up a methodology that is dedicated to collectively solve multi-agent mission planning that includes two components: path finding and task scheduling, and deal with complex environmental constraints and timing requirements of tasks.

In the field of robotics, design and development of GUI for planning, execution and supervision of missions involving several autonomous vehicles is getting increasingly much attention [40]. Such a GUI allows the operator of autonomous vehicles to plan and supervise several vehicles at the same time. Such user interfaces have been a research topic for different use cases including delivery services [41], military applications [42] and others [43, 44, 45]. Most of these GUI however, are designed for specific use cases and cannot be used as a generic graphical user interface for other domains. In this work we employ MMT which is a GUI that can be setup to communicate with different planners and autonomous vehicles. MMT has been used for planning and supervising underwater autonomous vehicles previously [10].

## 10.9 Conclusions and Future Work

In this article, we have presented a new methodology and a toolset to solve the mission planning problem of multiple autonomous agents. Our methodology includes an improved version of DALi for path planning, a timed-automata-based method for task scheduling, namely TAMAA, and an integration of these two methods to provide a complete solution of synthesis and verification of mission plans for multiple agents. As the improved version of DALi considers special road conditions, such as temporary obstacles, our method can deal with complex environments. TAMAA is based on formal modeling and model checking, so the synthesized task schedules are guaranteed to be the correct and the fastest to finish all tasks. The integration of DALi and TAMAA requires an iterative computation between path planning and task scheduling so that the result mission plans consider both the traveling time and task-execution time and are guaranteed to be the optimal solution. The methods have been implemented as a toolset named *MALTA*, which is made of three components. The front end of *MALTA* is a GUI for configuring the mission requirements and showing the results of mission planning. The back end of *MALTA*, which is responsible for running computational expensive functions, can be deployed locally or remotely. The middleware of *MALTA* bridges the front end and back end, so the users can focus on designing the map and tasks for the agents, and benefit from the algorithms of path planning and task scheduling without knowing the technical details. We have employed the toolset to solve a mission-planning problem of an industrial use case of an autonomous quarry. We have observed the computation time w.r.t. the numbers of obstacles, agents, heat areas, milestones, and the granularity of the map. The experimental results demonstrate the capability and limit of our method. An instantiated quarry is introduced and solved to show the flexibility of our method to fit various applications of mission planning.

There are two potential directions to extend our work in the future. One is to enrich the path-planning and task-scheduling algorithms supported by *MALTA*, so that the toolset can cope with more complex problems such as more agents or larger environments. To integrate the toolset with machine learning techniques is another direction. As the current task scheduling assumes the environment to be collaborative, it can be interesting to investigate how the method can be adapted when the environment contains some competitive agents.

## Acknowledgments

We acknowledge the support of the Swedish Knowledge Foundation via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr: 20150022, and via the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190-038.

# 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*. Springer, 1996.

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

[3] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report*, 1998.

[4] J. K. Lenstra. Job shop scheduling. In Mustafa Akgül, Horst W. Hamacher, and Süleyman Tüfekçi, editors, *Combinatorial Optimization*. Springer, 1992.

[5] Yasmina Abdeddaı, Eugene Asarin, Oded Maler, et al. Scheduling with timed automata. Elsevier, 2006.

[6] Alessio Colombo, Daniele Fontanelli, Axel Legay, Luigi Palopoli, and Sean Sedwards. Efficient customisable dynamic motion planning for assistive robots in complex human environments. *Journal of ambient intelligence and smart environments*, 2015.

[7] 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.

[8] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126, 1994.

[9] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. 2006.

[10] E Afshin Ameri, Baran Cürüklü, Branko Miloradovic, and Mikael Ektröm. Planning and supervising autonomous underwater vehicles through the mission management tool. In *Global Oceans 2020: Singapore–US Gulf Coast*, pages 1–7. IEEE, 2020.

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

[12] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*. Springer, 2003.

[13] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959.

[14] 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*, 1968.

[15] Peter Gjøl Jensen. Efficient analysis and synthesis of complex quantitative systems. 2018.

[16] Rong Gu, Peter Jensen, Danny Poulsen, Cristina Seceleanu, Eduard Paul Enoiu, and Kristina Lundqvist. Verifiable strategy synthesis for multiple autonomous agents: A scalable approach. *International Journal on Software Tools for Technology Transfer*, 2022.

[17] Rong Gu, Eduard Paul Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Verifiable and scalable mission-plan synthesis for multiple autonomous agents. In *25th International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2020.

[18] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[19] Nils J Nilsson et al. Shakey the robot. 1984.

[20] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 1996.

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

[22] Rachid Alami, Thierry Simeon, and Jean-Paul Laumond. A geometrical approach to planning manipulation tasks. the case of discrete placements and grasps. In *The fifth international symposium on Robotics research*. MIT Press, 1990.

[23] Kris Hauser and Jean-Claude Latombe. Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, 2010.

[24] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[25] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 2021.

[26] 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.

[27] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 1994.

[28] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 2009.

[29] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal methods for discrete-time dynamical systems*. Springer, 2017.

[30] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010.

[31] Mingyu Cai, Hao Peng, Zhijun Li, and Zhen Kan. Learning-based probabilistic ltl motion planning with environment and motion uncertainties. *IEEE Transactions on Automatic Control*, 2020.

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

[33] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 1998.

[34] Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In *International Conference on Computer Aided Verification*. Springer, 2010.

[35] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*. Springer, 2011.

[36] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis of reactive controllers for multi-agent systems. In *International Conference on Computer Aided Verification*, pages 251–269. Springer, 2016.

[37] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional and symbolic synthesis of reactive controllers for multi-agent systems. *Information and Computation*, 261:616–633, 2018.

[38] 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.

[39] 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.

[40] JW Eggers and Mark H Draper. Multi-uav control for tactical reconnaissance and close air support missions: operator perspectives and design challenges. In *Proc. NATO RTO Human Factors and Medicine Symp. HFM-135. NATO TRO, Neuilly-sur-Siene, CEDEX, Biarritz, France*, pages 2011–06, 2006.

[41] Khin Thida San, Sun Ju Mun, Yeong Hun Choe, and Yoon Seok Chang. Uav delivery monitoring system. In *MATEC Web of Conferences*. EDP Sciences, 2018.

[42] Youngjoo Kim, Wooyoung Jung, Chanho Kim, Seongheon Lee, Kihyeon Tahk, and Hyochoong Bang. Development of multiple unmanned aircraft system and flight experiment. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2015.

[43] Daniel Perez, Ivan Maza, Fernando Caballero, David Scarlatti, Enrique Casado, and Anibal Ollero. A ground control station for a multi-uav surveillance system. *Journal of Intelligent & Robotic Systems*, 2013.

[44] Bae Hyeon Lim, Jong Woo Kim, Seok Wun Ha, and Yong Ho Moon. Development of software platform for monitoring of multiple small uavs. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 2016.

[45] Espen Skjervold et al. Autonomous, cooperative uav operations using cots consumer drones and custom ground control station. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018.

# Chapter 11

# Paper D: Probabilistic Mission Planning and Analysis for Multi-agent Systems

Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist
Published in Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Springer, 2021.

## Abstract

Mission planning is one of the crucial problems in the design of autonomous Multi-Agent Systems (MAS), requiring the agents to calculate collision-free paths and efficiently schedule their tasks. The complexity of this problem greatly increases when the number of agents grows, as well as timing requirements and stochastic behavior of agents are considered. In this paper, we propose a novel method that integrates statistical model checking and reinforcement learning for mission planning within such context. Additionally, in order to synthesise mission plans that are statistically optimal, we employ hybrid automata to model the continuous movement of agents and moving obstacles, and estimate the possible delay of the agents' travelling time when facing unpredictable obstacles. We show the result of synthesising mission plans, analyze bottlenecks of the mission plans, and re-plan when pedestrians suddenly appear, by modelling and verifying a real industrial use case in UPPAAL SMC.

## 11.1 Introduction

Multi-Agent Systems (MAS) draw a wide interest in academia and industry, mostly due to their autonomous functions that ease people's daily lives and improve industrial productivity. Mission planning for MAS involves path planning and task scheduling, and is one of the most critical problems when designing such systems [1]. There are path-planning algorithms that have already proved useful for autonomous systems, e.g., RRT [2] and Theta* [3]. These algorithms are able to calculate collision-free paths towards a destination, yet they do not consider complex requirements and uncertainties in the environment. For instance, if agents need to prioritize or repetitively execute some tasks, path planning is not enough. In addition, when the task execution time is uncertain, or some moving objects such as humans and other machines appear irregularly in the environment, autonomous agents need to consider these factors when synthesising mission plans so that the resulting plans are comprehensive. Task scheduling algorithms are designed to solve the above problems. However, since task scheduling is an NP-hard problem, when the number of agents becomes large, traditional methods cannot manage to produce a result even for a simple instance with very restrictive constraints [4].

In our previous work, we have formally defined and modeled the movement and task execution of MAS [5], and proposed a combined model-checking and reinforcement learning method [6], to synthesise mission plans that are proved to satisfy complex requirements obtained from industry. However, when the agents perform some uncertain actions, e.g., unstable time of moving and operating, or the environment contains some stochastic phenomena, e.g., humans crossing the roads unpredictably, the proposed method does not provide quantitative verification and analysis, which is best suited in these cases.

In this paper, we propose an adjusted version of our method called MCRL (Model Checking + Reinforcement Learning) [6] to provide a means of synthesizing and analyzing mission plans for MAS with uncertainties of the type mentioned above. The method is based on Stochastic Timed Automata (STA) and statistical model checking (by employing UPPAAL SMC), and combines the latter with reinforcement learning. Instead of exhaustively exploring the state space of the model and looking for the execution traces that satisfy certain requirements, MCRL uses the simulation function of UPPAAL SMC to execute the model. Then, it adopts a reinforcement learning algorithm, namely Q-learning [7], to accumulate the rewards of the state-action pairs gathered in the simulation, and populate a Q-table that is used to guide the agents to move safely and finish tasks within a prescribed time limit. As the STA describe the stochastic behavior of the agents and uncertain events in the environment by probability distributions, based on which the simulation is executed, the collected state-action pairs reflect the possible scenarios that the agents would probably meet in the environment. Therefore, as long as the simulation generates enough data, the synthesised mission plans are comprehensive and optimal.

To estimate the possible delays of executing mission plans when the agents encounter unexpected situations, e.g., pedestrians, we adopt a hybrid-automata (HA) model of the agents that are equipped with a state-of-the-art collision-avoidance algorithm based on dipole flow fields [8]. By simulating and statistically verifying the HA model, we can get the estimated travelling time of the agents [9], respectively, which is then used to construct the STA model that is used for synthesising mission plans. Next, statistical verification and simulation of the STA are conducted in UPPAAL SMC in order to analyze the synthesised mission plans in an environment model containing uncertainties, which is not feasible by purely using reinforcement learning algorithms. To summarize, the contributions of this paper are:

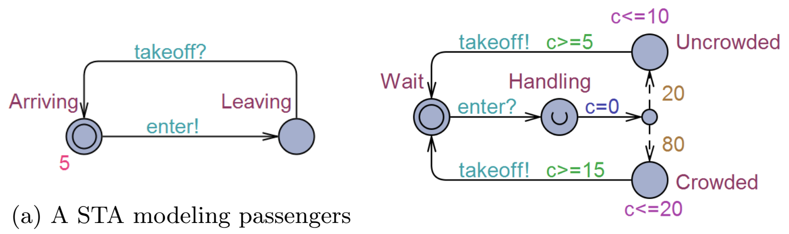
- An innovative approach based on MCRL for synthesizing and analyzing mission plans for MAS that exhibit stochastic behavior.
- An effective combination of the STA and HA models of MAS, which enables the estimation of travelling time considering unexpected situations, and thus produces comprehensive mission plans.
- An evaluation of the method showing the ability of analyzing the bottleneck of mission plans and re-planning when facing unpredictable moving obstacles.

The remainder of the paper is organized as follows. In Section 11.2, we introduce the preliminaries of this paper. Section 11.3 presents the problem and challenges. In Section 11.4, we introduce the adjusted version of MCRL and its combination with the HA model. Section 11.5 presents the bottleneck analysis as well as the ability of re-planning. In Section 11.6, we compare to related work, before concluding and outlining possible future work in Section 11.7.

## 11.2 Preliminaries

In this section, we introduce Stochastic Timed Automata and UPPAAL SMC, reinforcement learning, and a two-layer framework that we have proposed previously for formal modeling and verification of autonomous agents.

### 11.2.1 Stochastic Timed Automata and UPPAAL SMC



(a) A STA modeling passengers

(b) A STA modeling an airport

Figure 11.1: STA modeling a scenario of passengers arriving at an airport and taking off

UPPAAL SMC [10] is an extension of the tool UPPAAL [11], which supports Statistical Model Checking (SMC) of Stochastic Timed Automata (STA). STA is a widely used paradigm for modeling the probabilistic behavior of real-time systems. The basic elements of STA are locations and edges connecting them. Time can elapse at locations, which is reflected by the increased values of clock variables in delayed transitions of STA, whereas transitions between locations are non-delayed. The delays at locations follow probabilistic distributions, which are either uniform distributions for time-bounded delays, or exponential distributions (with user-defined rates) for unbounded delays. The choices between multiple enabled non-delayed transitions are also probabilistic.

Fig. 11.1 depicts a network of STA modeling the scenario of passengers arriving at an airport and taking off. Fig. 11.1a shows the model of passengers, who randomly arrive at the airport. The arriving time follows the exponential distribution as it is modeled by an unbounded delay at location *Arriving*. The constant "*5*" is the exponential rate that can be replaced by any rational number. The channels (e.g., *enter* and *takeoff*) model the handshaking interaction between STA. Note that

UPPAAL SMC only supports broadcast channels for a clean semantics of purely non-blocking automata. When a passenger enters an airport, the corresponding STA moves to location *Leaving* simultaneously with the airport STA (Fig. 11.1b) moving from location *Wait* to *Handling*, synchronized via the channel *enter*. Next, the airport STA goes to a branch point leading to two locations, namely *Crowded* and *Uncrowded*, respectively. The constants, "*20*" and "*80*", are the probability weights of the edges marked by the dashed lines in Fig. 11.1b, meaning that the probability of entering a crowded airport is *80%*, and *20%* for an uncrowded one. Delays at locations such as location *Crowded* are time-bounded, as the locations are constrained by invariants (e.g., $c <= 10$), so the delay time at these locations should not surpass the upper boundary specified by the invariants, respectively. If the outgoing edges of such locations are guarded by conditions, e.g., $c >= 5$ in our case, the STA cannot leave the locations until the lower boundaries of the guards are exceeded. A uniform distribution is set for the time-bounded delays by default in UPPAAL SMC, which is also adopted in this paper. Variables can be updated by assignments (e.g., $c = 0$) or C-code functions on the edges.

## 11.2.2 Reinforcement Learning

*Reinforcement learning* is a branch of machine learning that enables agents to learn how to take actions by themselves, in an environment. In this paper, we employ *Q-learning* [7] as the reinforcement learning algorithm to generate policies of movement and task execution for agents. 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:

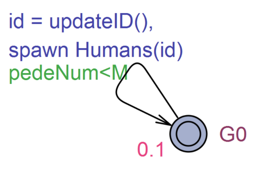
$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \underset{a'}{max} q^*(s',a')], \tag{11.1}$$

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 $s$, $\gamma$ is a constant of discounting, $s'$ is the new state coming from state $s$ by taking action $a$, $\underset{a'}{max}\ q^*(s',a')$ 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. The Bellman equation accumulates the Q-values of state-action pairs and guarantees the values to converge to the maximum Q-value during the learning process [12]. In this paper, we use the simulation function in UPPAAL SMC to gather the information of state-action pairs in files, and invoke a Java program to parse the data and run the Q-learning algorithm, so that a Q-table is populated.

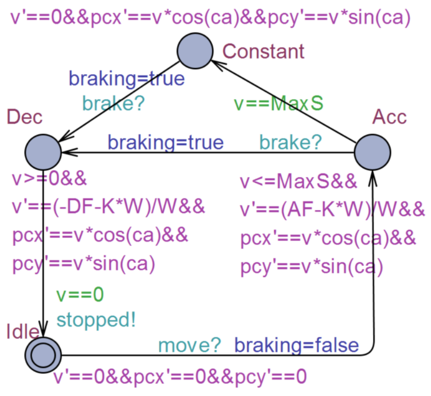
### 11.2.3 A Two-Layer Framework for Formal Modelling and Verification of Autonomous Agents

To provide a separation of concerns for the formal modeling and verification of autonomous agents, we have proposed a two-layer framework [9]. In this framework, a static layer is responsible for mission planning and only concerns static obstacles and milestones where the tasks are carried out. The dynamic layer uses hybrid automata (HA) [13] to model the continuous movement and operations of the agents in UPPAAL SMC. In addition, UPPAAL SMC provides a "spawning" function to dynamically generate instances of HA models during the verification, which enables one to mimic the sudden appearance of obstacles (e.g., pedestrians), which are considered unpredictable before the agents get close to them.

Fig. 11.2a shows the HA that generates pedestrians. As long as the number of pedestrians does not exceed a maximum number (i.e., "pedeNum<M"), the self-loop edge of location *G0* is enabled, which invokes the spawning function to generate an instance of the pedestrian model. The constant "*0.1*" denotes the rate of the exponential probability distribution of the pedestrians' appearance. Fig. 11.2b depicts the HA that models the continuous linear movement of agents. The model contains four locations, representing the four moving statuses of agents: idle, acceleration, constantly moving, and deceleration. At the each of the locations, the derivatives of speed and positions are regulated by Newtonian laws of motion in the form of ordinary differential equations (ODE). In a nutshell, the HA model describes the continuous movement of agents, and thus the simulation of the model reflects the agents' moving trajectories when circumventing obstacles. For brevity, we refer readers to the literature [9] for details. In this paper, we use this HA model to generate the moving trajectories of pedestrians and agents, and UPPAAL SMC to estimate the prolonged traveling time of the agents caused by collision avoidance, which is used for re-planning.



(a) An example of HA generating pedestrians



(b) An example of HA modeling the linear movement of agents

Figure 11.2: Examples of HA model in the dynamic layer of the framework

## 11.3 Problem Description

In this section, we introduce the research problem that originates from an industrial use case of an autonomous quarry, containing various autonomous vehicles, e.g., trucks, wheel loaders, etc. For example, as shown in Fig. 11.3, in an autonomous quarry, a wheel loader digs stones at stone piles and loads them into trucks, which carry the stones to a primary crusher, where stones are crushed into fractions, and proceed to carry the crushed stones to the secondary crushers, which is the destination. To accomplish their tasks and guarantee a certain level of productivity, these autonomous vehicles need to calculate collision-free paths and schedule their tasks (e.g., digging stones) to finish their jobs within a time frame. In this paper, henceforth, we name path planning and task scheduling as mission planning in general. As our solution is generic and suits all kinds of autonomous systems that need to synthesise mission plans, the autonomous vehicles in this paper are referred to as autonomous agents [14].

In this paper, path planning is accomplished by the Theta* algorithm [3] as the environment in the problem is a 2D map and the algorithm is especially good at generating smooth paths with any-angle turning

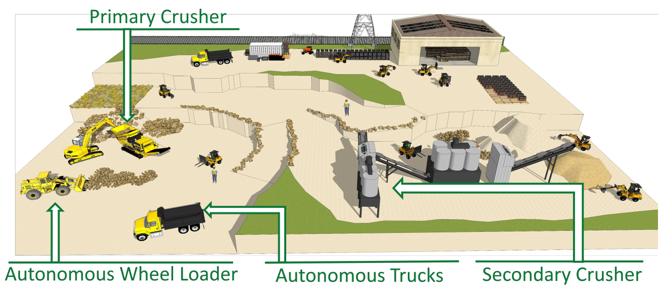


Figure 11.3: An example of an autonomous quarry

points in 2D maps. Task scheduling acquires satisfaction of various requirements, e.g., task assignment, execution order, and timing requirements. We extract the requirements of the autonomous quarry from our industrial partner, and generically categorize them as follows:

- *Task Assignment.* The task must be assigned to the right milestone containing the corresponding device.
- *Execution Order.* The task execution order must be correct, e.g., unloading into the primary crusher can start only after digging stones finishes.
- *Milestone Exclusion.* Some milestones containing a device that only allows one agent to operate at a time are exclusive when they are occupied.
- *Timing.* Tasks must be completed within a prescribed time frame.

The complexity of path planning of multiple agents increases linearly as the number of agents grows, because the path-planning algorithm runs on each individual agent and it does not consider the paths of other agents, as the collision avoidance is dealt with when the agents are actually moving. In other words, the time to calculate paths for multiple agents is the sum of the computation time of each agent. However, the task-scheduling problem is NP-hard and involves uncertainties that traditional methods do not consider [4].

- *Uncertain execution time of tasks.* The execution time of tasks is not a fixed value, but it is a time interval between the best-case execution time (BCET) and worst-case execution time (WCET), which are usually different.
- *Uncertain movement time.* Since some milestones are exclusive, when an agent approaches an occupied milestone, it most probably should wait until it is released. The waiting time is uncertain.
- *Uncertain environment.* Human workers sometimes appear in the sites but do not always stay there. This requires the agents to avoid those workers at all cost, and adjust their mission plans accordingly, in order to maintain productivity.

These features make our problem even more difficult than the classic scheduling problem. For example, if human workers appear irregularly, it is hard to estimate their influence on the traveling time of agents. We formulate the target problems of this paper as follows.
**Overall Challenge**. Given a confined environment containing multiple autonomous agents, several predefined milestones and static obstacles, some unpredictable moving objects or humans, a set of tasks for the agents to finish in order to satisfy some requirements, the goal is to synthesize mission plans for these agents, such that:

- The mission plans satisfy the requirements that are categorized previously;
- The mission plans consider the uncertainties in the environment and handle them effectively so that the agents could finish tasks under various conditions;
- The solution provides a means of statistical analysis of the synthesised mission plans to investigate the bottleneck of the plans, and an ability of re-planning when facing disturbance, e.g., pedestrians.

## 11.4 Mission Planning Based on Reinforcement Learning and Stochastic Timed Automata

In this section, we introduce the modelling of MAS using STA, which is based on a method called MCRL [6]. MCRL combines model checking and reinforcement learning, which enables the method to cope with large numbers of agents and verify the synthesised mission plans. The use of stochastic timed automata in this paper extends MCRL with the ability of modelling stochastic behaviors. We also present some queries that are used in this method for statistical analysis of the mission plans.

### 11.4.1 MCRL: Combining Model Checking and Reinforcement Learning for Mission Planning

Previously, we have presented the formal definitions of agent movement and task execution and the model-generation algorithms to generate Timed Automata (TA) for mission-plan synthesis [5]. This initial work provides a theoretical foundation and a tool called TAMAA, based on which a novel approach is designed to synthesise mission plans, namely MCRL.

**Overall Description of MCRL.**

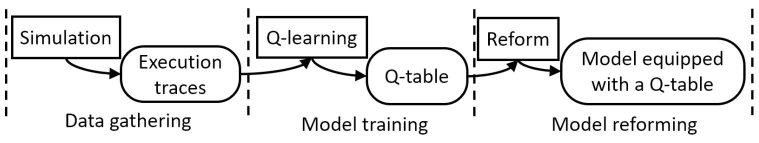


Figure 11.4: The process of the MCRL method

As Fig. 11.4 depicts, MCRL consists of three phases. First, it simulates the TA that models the movement and task execution of autonomous agents by running the Monte Carlo simulation query in UPPAAL SMC. The introduction of the TA model is in the literature [6]. The multi-round simulation produces the execution traces of the model. Some of them satisfy our requirements, e.g., finishing tasks in time, correct execution order of tasks; some traces fail, e.g., exceeding the time limit. The successful traces are assigned with positive values, which are calculated by $(ST - FT)^2$, where $ST$ is the simulation time, $FT$ is time of reaching the desired state, e.g., finishing all tasks; whereas a fixed negative value is assigned to all the failed traces.

Next, the traces and their values are input into the model-training phase, where a reinforcement learning algorithm, namely Q-learning, is performed to generate a Q-table. The Q-table contains the state-action pairs and their values that are accumulated by running Equation (11.1) using the data of the input traces. This equation guarantees that the values of state-action pairs converge, as long as the simulation has produced enough data of execution traces. Eventually, the Q-table is injected back to the TA model of agents, where a new TA named conductor is created so that the behavior of the agent model is controlled by it. The conductor TA looks up the Q-table and chooses the action that owns the highest value among the available actions at the current state for the agents to perform. Each agent model has its own conductor TA so that the agents can make decisions distributedly. However, as the Q-table contains the state-action pairs of all agents, when their actions conflict, e.g., moving to the same exclusive milestone simultaneously, the agents can compare their rewards of actions with others, and let the one having the highest reward to perform. In this way, the Q-table serves as the mission plan we intend to synthesise. In addition, since the method utilizes random simulation and reinforcement learning instead of pure exhaustive model checking, the solution is scalable for systems with large numbers of agents. For a detailed introduction of the method, we refer readers to the literature [6].

Although Q-learning strengthens MCRL's ability of handling large numbers of agents, the method provides no means of handling unpredictable events, which is important as the environment is uncertain. This limitation stems from the use of timed automata. This modelling language cannot depict the stochastic events in the environment. For example, when human workers sporadically appear in the environment, MCRL cannot estimate the possible delay that is caused by the detour taken by the agents to avoid humans. In addition, industries always focus on productivity. The waiting time of agents at exclusive milestones is an unnecessary consumption of time, but it is hard to capture as the waiting time depends on multiple factors. Original MCRL is not able to provide this kind of analysis, as it does not use any statistical analysing techniques.

### 11.4.2 Stochastic Timed Automata for MCRL

To overcome these shortcomings, we improve MCRL by adopting stochastic timed automata (STA) as the modelling language and statistical model checking for verification and analysis. In this section, we present the STA model in detail such that readers understand how the movement and task execution is modelled as STA, and how the stochastic behavior is handled by this model.

**STA of Task Execution.**

Tasks in this paper are operations of the agents that need to be carried out in a right order and at the specific milestones. For instance, in the scenario of an autonomous quarry in Fig. 11.3, tasks for autonomous trucks can be unloading stones into the primary crushers, charging, etc. Collaborative tasks are the ones that need more than one agent to perform, e.g., loading stones at stones piles needs a wheel loader and a truck to accomplish. For mission planning, a task can be abstracted as time duration between the BCET and WCET, which is only permitted to start when a set of conditions is satisfied, e.g., precedent tasks are finished, and staying at the right milestone. The formal definition of tasks is presented in literature [5].

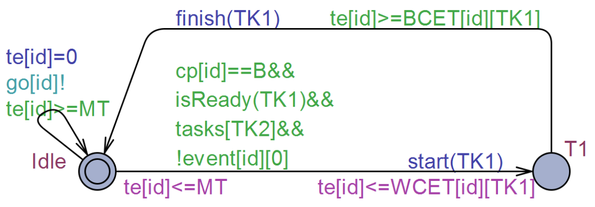


Figure 11.5: The STA modeling an agent executing task *T1*

Fig. 11.5 depicts an example of the STA modelling an agent executing one of its tasks, namely *T1*. For brevity, the execution of other tasks for the same agent, which should be modelled in the same STA, is not shown in this figure. Note that the variable *id* in this figure is the index of the agent. The STA starts from the location named *Idle* that represents the status of running no tasks. Agents are only allowed to move at this status, hence, this location has a self-loop edge labelled by a synchronization channel *go[id]* that is used to inform the movement

STA to start moving. Since the milestone that the agent is approaching to might be occupied and exclusive, the agent probably has to wait. The invariant on the location *Idle* (e.g., *te[id]<=MT*) and the guard on its self-loop edge (e.g., *te[id]>=MT*) is for triggering the "moving" command every *MT* time units, so that the agent would not wait forever and periodically detects whether the target milestone is available. The detection is done by the STA of agent movement, which is introduced in the next section.

If the agent decides to execute task *T1*, its task execution STA transfers to location *T1*. This edge is guarded by a Boolean expression that is composed of four parts (see Fig. 11.5). The first Boolean expression *cp[id]==B* checks if the agent is at milestone *B* currently, where the task is permitted. The following function *isReady(TK1)* returns a Boolean value indicating whether task *T1* is not finished yet. If *T1* is a collaborative task, this function also decides if the collaborating agents are ready for this task by checking if they are staying at the same right position, which is milestone *B* in this case. The Boolean array named *tasks* stores the execution status of tasks, namely finished or not, so *tasks[TK2]* here checks if the precedent task of *T1* is finished. The Boolean expression *!event[id][0]* indicates that the event monitored by this agent is not active, where the number "*0*" is the index of the event that can be replaced. An event can be a battery-level-low warning, or a critical-damage alert, etc., which needs to be prioritized than regular tasks, and responded within a time frame. The task execution time is between the BCET and WCET. Therefore, the invariant on location *T1* regulates that the clock variable should not exceed the WCET of *T1*, whereas the guard on the outgoing edge of this location decides the earliest time to leave this location to be later than the BCET. In UPPAAL SMC, the default probability distribution of time-bounded delays is uniform distribution. Hence, the execution time of task *T1* here is between the BCET and WCET with equal possibilities.

When the guards hold, agents can take the transition with the execution of function *start(TK1)* to start *T1*. This function changes the variable of the current task of the agent, and stores the current state of the agent, as well as the corresponding action taken at this moment into an array. The array, which represents the execution trace, will be printed by UPPAAL SMC in the end of the data gathering phase (see Fig. 11.4). The function *finish(TK1)* simply changes the variable of the current task to *Idle*, and checks if all the tasks have been finished when the agent should leave the environment and stop.

### STA of Agent Movement

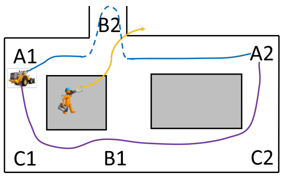
Fig. 11.6a depicts a scenario containing an intersection where pedestrians keep crossing the road every once a while. An autonomous vehicle starting from position *A1* intends to go to *A2*. Though going straightly to *A2* is the shortest path, potential collision avoidance might increase the travelling time, as shown by the blue trajectory. Therefore, the vehicle can alternatively choose to detour via position *B1*, as shown by the violet trajectory. As the HA described in Section 11.2.3 model the probable appearance of human workers and the continuous movement of agents equipped with a collision-avoidance algorithm based on dipole flow fields [8], we can verify the HA model against queries in the following forms in order to obtain the prolonged travelling time and its probabilities.

$$\texttt{Pr[<=T](<> arrived)} \tag{11.2}$$

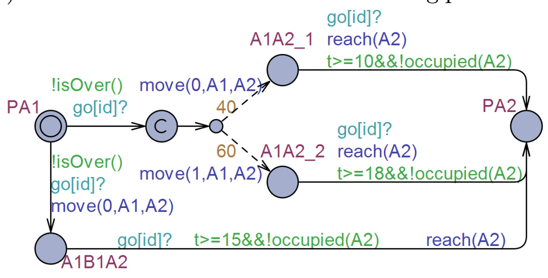
$$\texttt{Pr[<=T]([] arrived imply t <= TL),} \tag{11.3}$$

where `T` is the simulation time, `arrived` is a Boolean variable indicating if the agent arrives at the destination or not, `t` is a clock variable, and `TL` is an integer indicating the time limit. Query (11.2) calculates the probability of the agent reaching the destination, and Query (11.3) further calculates the probability of always arriving at the destination within `TL` time units. The results are probability intervals and we use the average value to estimate the probability of travelling time, which is used in the STA of movement.

Fig. 11.6b shows a part of the movement STA modelling the movement from *A1* to *A2*. As there are two alternative paths, the STA starts with a non-deterministic choice between two transitions to location *A1B1A2* or a branch point. The function *isOver()* returns a Boolean value of whether the agent has finished all tasks and should stop. The update function *move(0,A1,A2)* changes the current position of the agent, and stores the current state-action pair into the array, which is similar to the function *start()* in Fig. 11.5. When the agent chooses to go via position *B1*, which does not have any pedestrians, the STA transfers to the location *A1B1A2* representing the duration of travelling. When the least travelling time has passed, e.g., *15* time units travelling via *B1*, the STA can transfer to location *PA2*, as long as the milestone *A2* is not



(a) A scenario of an intersection containing pedestrians



(b) The STA modeling the possible movement of agents

Figure 11.6: A scenario of intersection and the STA modeling the movement of agents

occupied. If the travelling time is uncertain by the influence of pedestrians, the STA transfers to a branch point that leads to different locations representing different probable travelling duration, e.g., location `A1A2_1`. After verifying the HA of agents (see Figure 11.2 for an example) against queries similar to Queries (11.2) and (11.3), and replacing *TL* with different numbers, we can obtain that going to position *A*2 straightly can cost *10* or *18* time units, and their probabilities are 40% and 60%, respectively, which are depicted in Fig. 11.6b. In the STA of movement, a synchronization channel named *go[id]* is used to get commands from the task execution STA (Fig. 11.5). So the verification of agents is for an integrated model composing the STA of agent movement and task execution.

In UPPAAL SMC, a simulation query composed as following randomly executes the model for `R` rounds and `T` time units in each round,

```
simulate[<=T;R] {ds[0].cs,ds[0].act,ds[0].value,...}:tasks[TK1],
```
(11.4)

where `ds` is the array variable whose type is a structure, `cs` and `act` are the elements of the structure representing the current state and action, respectively, `value` is the reward or penalty assigned to the pair. The definitions of the states and actions are in the literature [6]. The predicate in the end of the query regulates that the data in the curly brackets are printed only when the predicate is true. In this query, when the agent finishes task *T1*, the elements in `ds` are printed. The simulation needs to run multiple runs for obtaining enough state-action pairs that simulate various situations that the agents would encounter. Hence, the Q-learning algorithm, which uses the state-action pairs as input, would cover various cases comprehensively so that the final mission plans can satisfy various properties in an environment model containing uncertainties.

### MCRL Revisited

Now that the TA of task execution and movement are adjusted to STA, the simulation query in UPPAAL SMC would explore the state space of the model based on the probability distributions defined in the STA. The model-training phase that uses the state-action pairs representing the stochastic behavior of agents would generate mission plans that are statistically optimal.

## 11.5 Statistical Verification and Analysis of the Use Case: an Autonomous Quarry

In this section, we evaluate our method by demonstrating a statistical verification and analysis on our use case: an autonomous quarry (as shown in Fig. 11.3). The experiments are conducted in UPPAAL 4.1.24. Most of the statistical parameters are set to the default values in UPPAAL SMC, except the probability of false negatives ($\alpha$), which is 0.001, and probability uncertainty ($\varepsilon$), which is 0.001. The experimental scenario is depicted in Fig. 11.7. Tasks for those agents are shown in Table 11.1. Milestones $A$ to $D$ are exclusive, thus only one truck is allowed at one time. As there are two primary crushers, the trucks need to choose one of them to perform tasks, which take uncertain execution time. The agents must carry all the stones to the secondary crusher, and the job need to be accomplished within a time frame.

Table 11.1: Tasks for the autonomous agents in the experiment

|  | Task | BCET | WCET | Precedent task | Milestone |
|---|---|---|---|---|---|
| Wheel loader | Dig | 2 | 2 | none | Stone pile (A) |
|  | Unload | 1 | 4 | Dig | Stone pile (A) |
| Truck | Load I | 1 | 4 | Dig | Stone pile (A) |
|  | Unload I | 4 | 4 | Load I | Primary crusher (B or C) |
|  | Load II | 2 | 3 | Unload I | Primary crusher (B or C) |
|  | Unload II | 3 | 5 | Load II | Secondary crusher (D) |

## 11.5.1 Mission Plan Synthesis

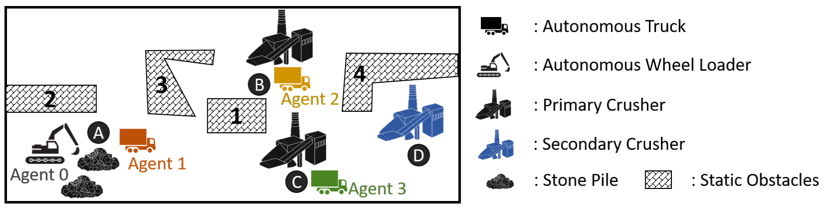


Figure 11.7: An experimental scenario containing 4 autonomous agents

After building the STA and running MCRL by using UPPAAL SMC and our Java program of the Q-learning algorithm, we successfully synthesize mission plans for agents. By verifying queries as following, we demonstrate the synthesized mission plans satisfy different kinds of requirements that are described in Section 11.3.

- *Task Assignment.* Query (11.5) checks the probability of agent $n$ performing task $T_i$ at milestone $P_i$. The results for all tasks in Table 11.1 are above 99.8%.

$$\texttt{Pr[<=T]([] te}_n\texttt{.T}_i\texttt{ imply m}_n\texttt{.P}_i\texttt{)} \tag{11.5}$$

- *Execution Order.* Query (11.6) checks the probability that when agent $n$ is performing task $T_i$, its precedent task $T_j$ has finished. UPPAAL SMC returns that the results for tasks that have precedent tasks are above 99.8%.

$$\texttt{Pr[<=T]([] te}_n\texttt{.T}_i\texttt{ imply te}_n\texttt{.tasks[j])} \tag{11.6}$$

- *Milestone Exclusion.* Query (11.7) checks the probability that when agent $n$ is at an exclusive milestone named $P_i$, other agents are not there. The results for milestones $A$ to $D$ are above 99.8%.

$$\texttt{Pr[<=T]([] m}_n\texttt{.P}_i\texttt{ imply !(m}_0\texttt{.P}_i\texttt{ \&\& ... \&\& m}_{n-1}\texttt{.P}_i\texttt{ \&\& m}_{n+1}\texttt{.P}_i\texttt{ ...))} \tag{11.7}$$

- *Timing.* Query (11.8) checks the probability of agent $n$ travelling through all milestones and finishing all tasks within $TL$ time units. If we set $TL$ to be *10* and *25* for wheel loaders and trucks, the results are above 99.8%.

$$\texttt{Pr[<=T]([] (te}_n\texttt{.tasks[0] \&\& ... \&\& te}_n\texttt{.tasks[M-1]) imply x < TL)} \tag{11.8}$$

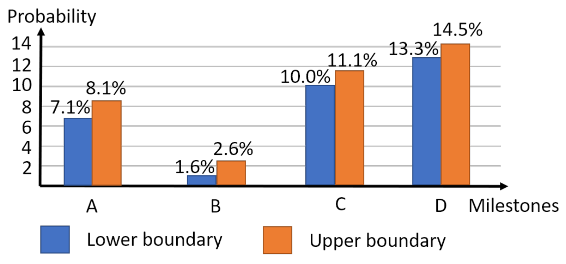
In these queries, $\texttt{te}_n$ and $\texttt{m}_n$ are the task execution STA and movement STA of agent $n$, respectively, $\texttt{te}_n\texttt{.tasts}$ is a Boolean array for storing the task execution status of agent $n$, namely *true* for finished tasks, and *false* for unfinished ones, `M` is the number of tasks, and `x` is a global clock variable that is only reset when all tasks finish.

## 11.5.2 Bottleneck Analysis

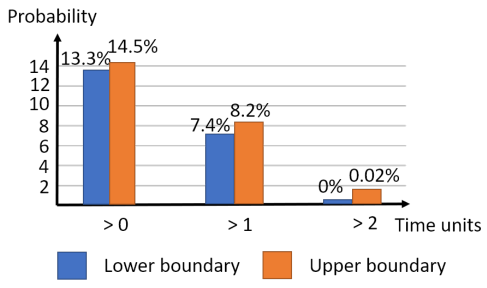
To perform this analysis, we verify the reformed model equipped with Q-tables against queries in the following form of Query (11.9) to get the waiting time at different milestones during the process of transferring stones.

$$\texttt{Pr[<=T](<> m0.wt[i] + m1.wt[i] + ... + mn.wt[i] > TL),} \tag{11.9}$$

where `T` is the simulation time, `m0` to `mn` are the movement STA of agents *0* to $n$, `wt[i]` refers to the waiting time at milestone $i$, and `TL` is an integer estimating the waiting time. By setting `TL` to zero and replacing the index `i` with the indices of milestones $A$ to $D$, one can investigate the probability of waiting at each milestone (see Fig. 11.8a). By replacing the integer `TL` with different values and fixing the index `i` to some certain milestone, one can estimate the waiting time at the milestone and the corresponding probability (see Fig. 11.8b). In UPPAAL SMC, the result of a probability estimation property (e.g., Query (11.9)) is given as a probability interval with a confidence level. Hence, the probabilities in Fig. 11.8 are presented as ranges from the lower boundaries to the upper boundaries. As shown in Fig. 11.8a, the probabilities of waiting at milestones $A$ to $D$ are always larger than zero, and the average probability of waiting at milestone $D$ is the highest. We specifically estimate the waiting time at milestone *D*. As shown in Fig. 11.8b, the waiting time is most likely less than 2 time units.



(a) Probabilities of waiting at milestones

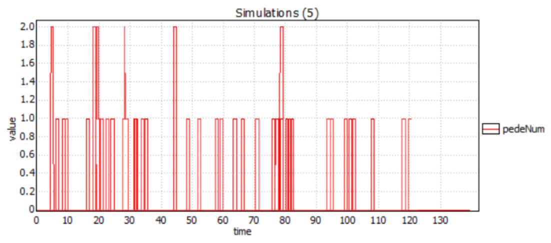


(b) Waiting time at milestone D

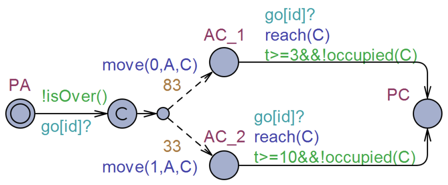
Figure 11.8: Bottleneck analysis of the scenario in Figure 11.7

### 11.5.3 Travelling Timed Estimation and Re-Planning

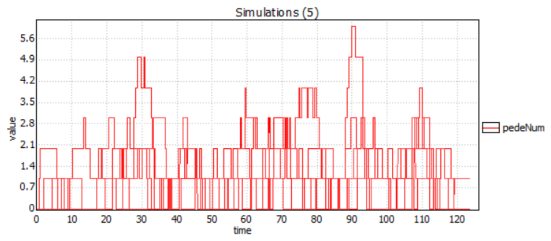
When the autonomous agents encounter pedestrians, they must run collision-avoidance algorithms to compute a new path to bypass the pedestrians, and that would possibly affect the travelling time significantly such that it is even quicker to take another path. We call the ability of agents choosing another path when encountering moving obstacles re-planning.



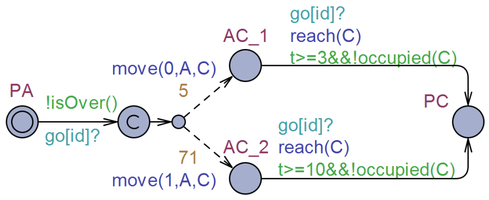
(a) The number of pedestrians. Exponential rate of the generator: 0.1. Existing time: 1



(b) The resulting movement STA in the situation with a few pedestrians



(c) The number of pedestrians. Exponential rate of the generator: 0.2. Existing time: 5



(d) The resulting movement STA in the situation with many pedestrians

Figure 11.9: The Number and frequency of pedestrians and the movement STA

In the scenario depicted in Fig. 11.7, if the number of autonomous trucks is decreased to one, the truck is free to choose between primary crushers at milestones $B$ and $C$, as no other trucks are competing with it. Since the primary crusher at milestone $C$ is closer to the secondary crusher, the Q-learning algorithm enables the autonomous truck to choose milestone $C$ rather than milestone $B$ as the precedent position of milestone $D$. We can verify this phenomenon by checking Query (11.10):

```
Pr[<=T] ([] m0.D imply (viaC && !viaB)),        (11.10)
```

where `viaC` and `viaB` are Boolean variables, which are turned to *true* when the agent sets off from the starting point, i.e., milestone $A$, and reaches milestones $C$ and $B$, respectively, and are turned back to *false* when the agent leaves milestone $D$. Hence, Query (11.10) checks the probability of an agent going to location `D` via location `C` but not location `B`.

However, if pedestrians keep walking near milestone $C$ and thus block the path (see Fig. 11.7), it could take longer time if the agent sticks to the original path plan (i.e., travelling via milestone C). By using the HA model depicted in Fig. 11.2a, we can generate instances of the pedestrian model dynamically during verification. Then we verify the HA model that describes the continuous movement of agents (see Fig. 11.2b for an example of linear movement) together with the pedestrian model against queries in the form of Queries (11.2) and (11.3), in order to estimate the prolonged travelling time between milestones $A$ and $C$, and the corresponding probabilities. Next, we encode the new travelling time and its probabilities into the movement STA and synthesize mission plans.

Fig. 11.9 shows two situations of the scenario, where pedestrians are few and crossing the road quickly (Fig. 11.9a), as well as pedestrians are many and walking slowly (Fig. 11.9c), which causes congestion on the road. The situation with fewer pedestrians results in the movement STA that is partly shown in Fig. 11.9b, where the probability of going to milestone $C$ quickly is 83% (i.e., $t >= 3$), whereas 33% is the probability of moving slowly (i.e., $t >= 10$). Similarly, the situation containing many pedestrians results in the movement STA partly depicted in Fig. 11.9d, where the chance of agents moving slowly is much larger than the chance of moving quickly.

Verifying Query (11.10) against the model that is partly shown in Fig. 11.9d produces a result of a range of low probabilities, where as if query is changed to check the probability of agents going via milestone $B$, the result is much higher. This shows that MCRL enables the agents to re-plan a better path when the irregular appearance of pedestrians influences the path plans.

## 11.6 Related Work

Motion-plan synthesis has arisen a wide interest of research in recent years. Nikou et al. [15] present a method of automatic controller synthesis for multi-agent systems under the presence of uncertainties. Sadraddini et al. [16] propose an approach of synthesising control strategies for positive and monotone systems, which satisfy requirements formalized by Signal Temporal Logic, and demonstrate their method on a traffic management case study. Wang et al. [17] propose a novel formulation based on Partially Observable Markov Decision Processes to synthesis policies over a vast space of probability distributions. Although having promising results, these methods are not applied in industrial systems, which requires solutions to be practically usable and scalable.

To model the uncertain behavior of the autonomous agents and environment, Markov Decision Process and Probabilistic Computation Tree Logic (PCTL) have been adopted by many studies. A solution of behavior verification of autonomous vehicles (AV) proposed by Sekizawa et al. [18] considers the disturbance that causes the AV to swerve from the planned path. Their solution uses the probabilistic model checker PRISM to conduct the verification against PCTL properties. Al-Nuaimi et al. [19] also employs PRISM in their design of a stochastically verifiable decision making framework for AV. The authors demonstrate the applicability of their framework in a scenario of parking bay containing one AV, a pedestrian, and another vehicle. Ayala et al. [20] present a solution to find control strategies for mobile robotic systems moving in environments containing entities that are not completely observable. Compared with these studies, our approach systematically estimates the disturbance caused by unpredictable moving obstacles, and enables replanning for the autonomous agents. UPPAAL STRATEGO is designed to synthesize strategies for stochastic priced timed games [21], and it also implements the Q-learning algorithm as one of its algorithms for synthesis. The main difference between MCRL and UPPAAL STRATEGO is that the former supports a larger numbers of agents, and we refer the interested readers to previous work [6] for a detailed comparison between the methods.

## 11.7 Conclusions and Future Work

We present a method for automatic synthesis of mission plans for multi-agent systems. The method is based on MCRL, which combines model checking with reinforcement learning, and extends MCRL with the ability of handling uncertainties in the environment by employing Stochastic Timed Automata and Statistical Model Checking. We demonstrate the applicability of the method in an industrial use case: an autonomous quarry, provided by VOLVO CE. The demonstration shows that the method is capable of synthesising mission plans for MAS that satisfy various requirements, and further analyse the bottleneck of the mission plans. When encountering disturbance of unpredictable moving obstacles, e.g., pedestrians, the method is able to estimate the delays of traveling time of the agents, and conduct a re-planning when it is necessary. Future work includes integrating the new MCRL with our tool called TAMAA [5], so that a complete solution of mission-plan synthesis for MAS together with a user-friendly GUI is accomplished. Automating the transformation of requirements into temporal logic queries is another possible direction.

## Acknowledgments

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] 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.

[2] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Computer Science Dept., Iowa State University, 10 1998.

[3] 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.

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

[5] Rong Gu, Eduard Paul Enoiu, and Cristina Seceleanu. Tamaa: Uppaal-based mission planning for autonomous agents. In *35th ACM/SIGAPP Symposium On Applied Computing SAC2020*. ACM, 2019.

[6] Rong Gu, Eduard Paul Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Verifiable and scalable mission-plan synthesis for multiple autonomous agents. In *25th International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2020.

[7] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989. King's College, Cambridge.

[8] 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. Frontiers Media SA.

[9] 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.

[10] 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.

[11] 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. Springer.

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

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

[14] 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.

[15] Alexandros Nikou, Jana Tumova, and Dimos V Dimarogonas. Probabilistic plan synthesis for coupled multi-agent systems. *IFAC-PapersOnLine*, 50(1):10766–10771, 2017. Elsevier.

[16] Sadra Sadraddini and Calin Belta. Formal synthesis of control strategies for positive monotone systems. *IEEE Transactions on Automatic Control*, 64(2):480–495, 2018. IEEE.

[17] 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, ACM, 2018.

[18] Toshifusa Sekizawa, Fumiya Otsuki, Kazuki Ito, and Kozo Okano. Behavior verification of autonomous robot vehicle in consideration of errors and disturbances. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 550–555. IEEE, 2015.

[19] Mohammed Al-Nuaimi, Hongyang Qu, and Sandor M Veres. A stochastically verifiable decision making framework for autonomous ground vehicles. In *2018 IEEE International Conference on Intelligence and Safety for Robotics (ISR)*, pages 26–33. IEEE, 2018.

[20] AI Medina Ayala, Sean B Andersson, and Calin Belta. Temporal logic control in dynamic environments with probabilistic satisfaction guarantees. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3108–3113. IEEE, 2011.

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

# Chapter 12

# Paper E: Correctness-Guaranteed Strategy Synthesis and Compression for Multi-Agent Autonomous Systems

Rong Gu, Peter G. Jensen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist

Submitted to Journal of Science of Computer Programming, Elsevier, 2022.

## Abstract

Planning is a critical function of multi-agent autonomous systems, which includes path finding and task scheduling. Exhaustive search-based methods such as model checking and algorithmic game theory can solve simple instances of multi-agent planning. However, these methods suffer from state-space explosion when the number of agents is large. Learning-based methods can alleviate this problem, but lack a guarantee of correctness of the results. In this paper, we introduce MoCReL, a new version of our previously proposed method that combines model checking with reinforcement learning in solving the planning problem. The approach takes advantage of reinforcement learning to synthesize path plans and task schedules for large numbers of autonomous agents, and of model checking to verify the correctness of the synthesized strategies. Further, MoCReL can compress large strategies into smaller ones that have down to 0.05% of the original sizes, while preserving their correctness, which we show in this paper. MoCReL is integrated into a new version of UPPAAL STRATEGO that supports calling external libraries when running learning and verification of timed games models.

## 12.1 Introduction

Autonomous agents (or shortly, agents), such as driverless cars, drones, and mobile robots, are systems that can move, carry out tasks, and collaborate with other agents autonomously without human intervention. *Multi-Agent Autonomous Systems* (MAS) [1] consist of multiple agents that work together in an environment and aim to achieve a common goal, an example being a group of construction equipment quarrying, crushing, and transporting stones. Planning for MAS involves *path finding* and *task scheduling*, and is one of the most critical problems when designing such systems [2]. There exist algorithms that solve each problem, respectively. A* [3] and rapidly-exploring random tree (RRT) [4] are two well-known algorithms that calculate the shortest paths in an environment with static obstacles. Algorithms for task scheduling have also been widely researched, resulting in search-based methods [5, 6] and learning-based methods [7, 8].

Nevertheless, approaches that solve the entire planning problem for MAS, which also provide a correctness guarantee are often not scalable [9, 10]. Learning-based methods address this weakness but fail to provide a formal guarantee of the correctness of their results. A united solution that solves both path finding and task scheduling is still missing. The difficulties of finding such a solution are threefold. *First*, the tasks of the agents are of different kinds. Some must be done individually, whereas some need collaborations, that is, agents gather at the same position and start and finish a common task simultaneously. In addition, tasks have uncertain completion time, which increases the difficulty of task scheduling dramatically. *Second*, tasks can be scheduled differently: periodically (repeatedly perform A), sequentially (perform A, then B, then C), or as a request-response pair (whenever A occurs, perform B). *Third*, the complexity of solving the problem increases exponentially when the number of agents increases linearly. This difficulty stems from the fact that task scheduling is NP-hard [11]. Solving the problem algorithmically on MAS resulting from composing all agents' behaviors is computationally demanding.

We have previously proposed MCRL (Model Checking + Reinforcement Learning) [12, 13] as a method that combines model checking with reinforcement learning to synthesize and verify plans of agents. MCRL benefits from both model checking and reinforcement learning so that the scale of the problem that MCRL can solve is larger than that of search-based methods, and also the results (a.k.a., plans) are guaranteed to be correct by model checking. However, MCRL has some limitations: (i) models are hard to build manually when the environment is big or the agents are many; (ii) MCRL only supports simple tasks that are executed individually and periodically; (iii) the resulting plan synthesized by MCRL is larger than needed, as it contains a tabulation of system states that are unreachable under the plan, which is impeding understandability (by an operator) and the realizability on systems with limited resources.

To alleviate these issues, we propose *MoCReL* (Model-checked Compressed Reinforcement Learning). MoCReL provides functions of synthesizing, verifying, and compressing plans, and it relies on modeling MAS as *(Stochastic) Timed Games* in UPPAAL STRATEGO [14], which is a tool that incorporates a symbolic model checker UPPAAL [15], a statistical model checker UPPAAL SMC [16], a solver for *Timed Games* UPPAAL TIGA [17], and solvers for *Stochastic Timed Games* relying on learning algorithms [14]. Similar to MCRL, the plan synthesis in MoCReL is an iterative process of a random simulation and reinforcement learning. The simulation explores the MAS model randomly and samples the model's execution traces that record the executed *action* at each *state* of the model. Then the learning algorithm uses these traces to synthesize a plan, which is used in the next round of simulation. This iteration ends with a final plan generated until reaching the maximum rounds of iteration, or a user-defined number of traces are sampled. Next, to guarantee the correctness of the plan, MoCReL verifies it by model checking the MAS model under the control of the plan, that is, the plan controls the model to choose certain actions at different states. The selected pairs of state and action are labeled during the verification, which in turn helps compressing the plans. The unlabeled pairs are considered useless for satisfying the requirements, and thus are removed from the plan. In this way, plans are compressed while preserving the satisfied requirements. All the activities of plan synthesis, verification, and compression are implemented as an external library that is linked to UPPAAL STRATEGO, which enables us to easily change or extend the algorithms for learning and compression.

In summary, MoCReL overcomes the limitations of MCRL as follows, which are the contributions of this paper:

(i) Models in MoCReL are instances of templates, which facilitates automatic model construction. In our experiments (Section 12.5), we design and use a tool that is capable of generating the models based on the templates.

(ii) The model templates also allow for more task types, such as collaborations among agents and tasks that are activated by events.

(iii) MoCReL's method for plan synthesis and compression is proven to be sound, that is, plans that are synthesized and compressed by MoCReL must be correct.

(iv) Experiments of MoCReL on a real industrial case study shows that the compressed plans can take down to 0.05% of the memory space of the original plans, while preserving their properties, e.g., always eventually finishing all tasks.

The remainder of the paper is organized as follows. In Section 12.2, we introduce the preliminaries: timed games and strategies in UPPAAL STRATEGO, and reinforcement learning. Section 12.3 describes the problem of MAS planning. In Section 12.4, we describe our proposed methods for strategy synthesis, verification, and compression in MoCReL. Next, we present the experimental evaluation in Section 12.5. In Section 12.6, we compare to related work, and conclude the paper in Section 12.7, where we also mention directions for future work.

## 12.2 Preliminaries

In this section, we recall the timed automata formalism as used in the UPPAAL tool suite, timed games, and the reinforcement learning algorithm used in this paper. We denote non-negative integers as $\mathbb{N}$, and real numbers as $\mathbb{R}$.

### 12.2.1 UPPAAL Timed Automata

A *timed automaton* (TA) is finite-state automaton extended with real-valued variables [18]. The variables model the logic clocks in systems, which are zero initially and then increase synchronously with the same rate. UPPAAL [15] is a tool for modeling, simulation, and model checking of UPPAAL *timed automata* (UTA), which is an extension of TA with data variables, etc. A UTA is defined as a tuple:

$$< L, l_0, \Sigma, V, C, E, I >, \tag{12.1}$$

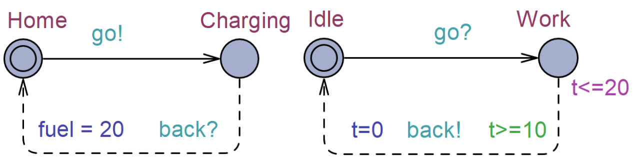
where $L$ is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $\Sigma$ is a set of *actions*, $V$ is a set of *data* variables, $C$ is a set of real-valued variables called *clocks*, $E \subseteq L \times B(C,V) \times \Sigma \times 2^C \times L$ is the set of *edges*, where $B(C,V)$ 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 $I : L \mapsto B(C)$ is a function assigning *invariants* to locations.

The semantics of a UTA is defined as a *timed transition system* over states $q = (l,\ c)$, where $l$ is a location, $c \in \mathbb{R}^C$ is the valuations of the clocks at this location, with the initial state $q_0 = (l_0, c_0)$, where $c_0$ assigns all clocks in $C$ to zero. There are two kinds of transitions:

(i) *delay transitions*: $q_n \xrightarrow{d} q'_n$, where $n \in \mathbb{N}$, $c \models I(l)$, $q'_n = (l, c \oplus d)$ is the next state delaying from $q_n$, and $c \oplus d$ is obtained by incrementing all clocks with the delay amount $d$ such that $c \oplus d \models I(l)$, and

(ii) *discrete transitions*: $q_n \xrightarrow{a} q_{n+1}$, where $q_{n+1} = (l', c')$ is the next state traversing via the edge $l \xrightarrow{g,a,r} l'$ from $q_n$, for which the guard $g$ evaluates to *true* in the source state $q_n$, $a \in \Sigma$ is an action, and valuation of $c'$ on the target state $q_{n+1}$ are obtained by resetting all clocks in $r \subseteq C$ such that $c' \models I(l')$.

### 12.2.2 Timed Games



(a) A TG of a car (b) A TG of a charging station

Figure 12.1: An example of a network of TG.

A *timed game* (TG) is a UTA with its set of actions partitioned into *controllable* ($\Sigma_c$) and *uncontrollable* ($\Sigma_u$) ones. UPPAAL STRATEGO [14] is a tool that supports modeling and verifying TG as well as synthesizing strategies to solve TG. Fig. 12.1 depicts two templates of TG in

UPPAAL STRATEGO, which consist of *locations* and *edges*. A template may also have local declarations and parameters and can be instantiated by a process assignment (in the system definition) [15]. In a TG template, locations (e.g., `Charging`) are blue circles. The double circles (e.g., `Home`) denote the initial location. *Clocks* (e.g., `t`) are special variables that increase simultaneously at rate 1, when the TG is executed. *Invariants* (e.g., `t<=20`) on locations must be *true* when the TG stays at the location. *Edges* connecting locations denote *discrete actions*, which are partitioned into *controllable* ones (solid lines) and *uncontrollable* ones (dashed lines). *Delays* allow time to elapse on locations as long as the associated invariants are not violated. *Guards* (e.g., `t>=10`) on edges must be *true* when the edges are enabled for transition. *Assignments* on edges reset clocks (e.g., `t=0`) or update data variables (e.g., `fuel = 20`). A *network* of TG is a parallel composition of TG that can synchronize via *channels* (e.g., `go!` is synchronized with `go?`).

When TG are executed, the choices of delaying at locations or executing discrete actions are non-deterministic, whereas *Stochastic Timed Games* (STG) replace the non-deterministic choices with stochastic ones. By default, STG in UPPAAL STRATEGO apply uniform probability distributions on discrete transitions and time-bounded delays, and exponential probability distributions on unbounded delays.

In this paper, we denote TG (STG) by $\mathcal{G}$ ($\mathcal{P}$), and the semantics of a $\mathcal{G}$ by $S_\mathcal{G}$. A run $\pi$ of a $\mathcal{G}$ is a sequence of alternating delays (denoted by $d$) and discrete transitions (denoted by $a$) of its $S_\mathcal{G}$: $\pi = q_0 \xrightarrow{d_1} q'_0 \xrightarrow{a1} q_1 \xrightarrow{d_2} q'_1 \xrightarrow{a2} ... \xrightarrow{d_n} q'_{n-1} \xrightarrow{a_n} q_n ....$ If we denote the last state of a finite run $\pi_f$ as $last(\pi_f)$, a *strategy* is a function that maps actions, i.e., either a controllable one $a \in \Sigma_c$ or a delay (indicated by the symbol $\lambda$), to each of the states. Formally, strategies are defined as follows [19]:

**Definition 13** (Strategy). *Let* $\mathcal{G} =< L, l_0, \Sigma_c \cap \Sigma_u, V, C, E, I >$ *be a TG. A strategy* $\sigma$ *over* $\mathcal{G}$ *is a partial function:* $\pi_f \to 2^{\Sigma_c \cup \{\lambda\}} \setminus \{\emptyset\}$ *such that for any finite run* $\pi_f$ *ending in state* $q_l = last(\pi_f)$*, if* $a \in \sigma(\pi_f) \cap \Sigma_c$*, then there must exist a transition* $q_l \xrightarrow{a} q_{l+1} \in S_\mathcal{G}$*.* □

A *stochastic* strategy of an STG delivers probabilities instead of definite choices of actions [19]. If we denote the set of runs in $S_\mathcal{G}$ as $\Pi_\mathcal{G}$, a TG under the control of a strategy $\sigma$ as $\mathcal{G} \mid \sigma$, the outcome of running $\mathcal{G} \mid \sigma$ is a subset of $\Pi_\mathcal{G}$, denoted as $Out(\mathcal{G} \mid \sigma)$. $Out(\mathcal{G} \mid \sigma)$ can be defined inductively as follows[1]:

**Definition 14** (Outcome of $\mathcal{G} \mid \sigma$)**.** *Given $q_0 \in Out(\mathcal{G} \mid \sigma)$, if $\pi \in Out(\mathcal{G} \mid \sigma)$ then $\pi' = last(\pi) \xrightarrow{e} q$ and $\pi' \in Out(\mathcal{G} \mid \sigma)$ if either one of the following conditions hold:*

1. *$e \in \Sigma_u$, or*
2. *$e \in \Sigma_c$ and $e \in \sigma(last(\pi))$, or*
3. *$e \in [0, T] \subseteq \mathbb{R}_{\geq 0}$ and $\forall e' < e$, $last(\pi) \xrightarrow{e'} q'$ for some $q'$ s.t. $\sigma(q') \ni \lambda$, where $T$ is the invariant boundary on the location of $last(\pi)$.* □

We will use these three conditions in the proof of Theorem 1. Let $P$ be a proposition and the reachability objective for $\mathcal{G}$, a finite run $\pi_f$ is *winning* w.r.t. $P$, if $P$ is true at the last state of $\pi_f$. A strategy $\sigma$ over a $\mathcal{G}$ is winning if all runs in $Out(G|\sigma)$ are winning. A *memoryless* strategy makes decisions on actions depending on the current state only, that is, a function $\sigma$: $last(\pi_f) \to 2^{\Sigma_c \cup \{\lambda\}} \setminus \{\emptyset\}$. In this paper, we aim to synthesize *winning*, *memoryless*, and *non-lazy* strategies, that is, winning strategies that urgently decide on a controllable action or *wait* until the environment makes a move[2]. Strategies referred to in the rest of this paper are all *memoryless* and *non-lazy.*

### 12.2.3 Model Checking and Temporal Properties

Model checking [20] traverses the state space of a formal model (e.g., UTA) and checks if it satisfies certain properties. The properties in this paper are of the following forms, where $p$ is an atomic proposition over the locations, clocks, and data variables of the UTA:

(i) **Invariance**: $E$`[]` $p$ meaning that there exists a run where all the states satisfy $p$, or $A$`[]` $p$ meaning that for all runs, $p$ is satisfied by all states in each run,

(ii) **Liveness**: $A$`<>` $p$ meaning that for all runs, $p$ is satisfied by at least one state in each run.

[1]Definition 14 is adapted from the definition strategy outcome in the literature [19].

[2]Memoryless and non-lazy strategies are shown to suffice for optimal scheduling of Duration Probabilistic Automata [5].

### 12.2.4 Reinforcement Learning

*Reinforcement learning* (RL) [21] is a kind of machine learning method for training agents by assigning rewards to desired behaviors and/or penalties to undesired ones, with the purpose of maximizing the accumulated rewards. Model-free RL relies on samples from the environment, which can be a virtual or a real one, to estimate the rewards of the future state-action pairs following the agent's current state. Model-based RL uses the model's predictions or distributions of state-action pairs and their rewards to find optimal actions. Therefore, models in the model-based RL must contain the full information of the environment and agents, which is hardly to obtain in an unexplored or partially observed environment.

*Q-learning* [22] is one of the model-free algorithms, which, at the limit, converges to the *optimal policy* for agents. Policies are associated with a state-action value function called *Q function.* The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \underset{a'}{MAX}\ q^*(s',a')], \tag{12.2}$$

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 $s$, $\gamma$ is a discounting value, $s'$ is the new state coming from state $s$ by taking action $a$, and $\underset{a'}{MAX}\ q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair $(s',a')$. The Bellman equation calculates the rewards of state-action pairs by considering both the current reward and the discounted maximum future reward. The rewards of the pairs are often stored in a score table. We show an example of such score tables in Section 12.3.2.

## 12.3 Problem Description

In this section, we introduce the planning problem of MAS and its challenges.

### 12.3.1 Overall Description

MAS are designed to move and execute a series of tasks autonomously. The actions belonging to a MAS can be categorized as: (i) movement, and (ii) executing a task. Whenever an agent moves or starts a task, the environment decides the ending time of the action. Now, the MAS planning is to order these two kinds of actions such that the MAS can finish its tasks while satisfying certain requirements, e.g., never let two agents execute the same task simultaneously, no matter how the environment reacts. The overall goal of MAS planning is formulated as below:
**Overall Goal**: Given a MAS and a set of requirements, the goal of planning is to order the agents' actions of movement and task execution, according to their variable ending time and occurrences of events, which are decided by the environment, such that the MAS can finish its tasks and satisfy the requirements.

**Remark 6.** *One can reduce the planning problem to a path-finding problem by removing the actions of task execution, or a task-scheduling problem by removing the actions of movement. Our algorithm is capable of solving the general problem that contains path finding and task scheduling or only one of them.*

**Remark 7.** *The requirements can be functional ones, such as always start task A after task B finishes, and safety ones, such as no collision with the static obstacles in the environment happens.*

### 12.3.2 Challenges of Solving the Planning Problem

The major challenges of this problem stem from four aspects, which get amplified especially when solving the problem via algorithmic techniques:

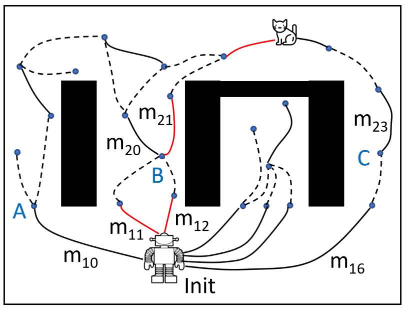
- **Challenge I** (uncertainty): The agents' actions have uncertain execution time, which means agents can choose actions to perform but cannot control how long time the actions will take. The uncertainty of execution time makes static plans inefficient, since they assign starting time to the actions without knowing their actual ending time.
- **Challenge II** (variety of task constraints): Some tasks have additional constraints, e.g., task A should always be completed before task B starts. Some tasks must be executed whenever certain events occur.

- **Challenge III** (complexity): As an NP-hard problem [11], when synthesizing and verifying plans for MAS, the state space of the model grows exponentially when the number of agents increases linearly as shown in the literature [9, 10].
- **Challenge IV** (large plans): As the state space of the problem grows exponentially, the resulting plan can grow exponentially too. However, some of the information in the plans may never be used. It is time-consuming to look for the right actions in a large plan. In some applications, it is simply impossible to store plans that take too much memory space, such as in Airborne Collision Avoidance System X case [23].

To give a concrete example of large plans, in Fig. 12.2, we show a path-finding problem in a 2D space, where a robot tries to catch a cat. Note that our mission-planning problem combines path finding and task scheduling, which makes the model state space to be high dimensional rather than a 2D space.

Algorithmic planning methods, such as the Dijkstra's algorithm for path finding [24], and the symbolic on-the-fly algorithm for solving timed games [17], usually explore the model's state space in a certain order (e.g., depth-first exploration), store the preceding states of each state, and back propagate to the initial state when finding the goal state. The resulting plan is *concise* as it only contains the state-action pairs that are *correct*, that is, they satisfy the requirements and reach the goal state. Additionally, the correctness of the plan is guaranteed as the algorithms explore the state space exhaustively [17]. However, the algorithmic methods are not scalable and when the model's state space becomes large, they fail to solve the problem in a reasonable time [13].

A path-finding algorithm that uses reinforcement learning can alleviate this problem by replacing the exhaustive state-space exploration with random simulation [13], while in turn suffering from disadvantages that we emphasize in the following. As depicted in Fig. 12.2a, a path plan synthesized by reinforcement learning possibly explores multiple routes from the robot to the cat, and results in a score table shown in Fig. 12.2b. The score table only contains controllable actions. A robot under the control of a plan always chooses the actions with the highest score at each of its states. For example, initially, the robot non-deterministically chooses one action among $\mathtt{m}_{11}$ and $\mathtt{m}_{12}$, because they have the highest score at the state `Init`. These scores are accumulated gradually during



(a) An example of a plan for a path-finding problem. Solid lines are controllable actions in the plan. Dashed lines are the uncontrollable actions of the environment. Red lines are the ones that guarantee to reach the destination no matter how the environment reacts. Black lines are useless ones that should be removed from the plan.

| | | | | | |
|---|---|---|---|---|---|
| | $m_{10}$ | $m_{11}$ | $m_{12}$ | ... | $m_{16}$ |
| Init | -2 | 3 | 3 | ... | 1 |
| | | | ... | | |
| | $m_{20}$ | $m_{21}$ | | | |
| B | 1 | 4 | | | |
| | $m_{23}$ | | | | |
| C | 2 | | | | |
| | | | ... | | |

(b) The score table of the plan for this example. The first column indicates states. The grey rows indicate the controllable actions at states. The white rows show the scores of state-action pairs accumulated by reinforcement learning.

Figure 12.2: An example of path finding in an environment with uncertain behaviors and the score table of the path plan.

the course of learning, hence, although the score of the pair ($\texttt{Init}$, $\texttt{m}_{10}$) ends to be $-2$, one cannot neglect it before the learning finishes. Another example of useless data is the pair ($\texttt{C}$, $\texttt{m}_{23}$). It is sampled during the random simulation, but not used in the final plan, which initially chooses to do actions $\texttt{m}_{11}$ and $\texttt{m}_{12}$, and thus never gets to state $\texttt{C}$. Besides, as the synthesis is based on random simulations, there is no guarantee on the correctness of the results, that is, the actions with the highest score are not guaranteed to lead the agents towards the goal state and satisfy all requirements. Hence, there is a need of removing the useless data from the plan while guaranteeing the correctness of the result.

**Overall challenge**. In a nutshell, the overall challenge of MAS planning is to design a method for plan synthesis that can cope with the uncertain execution time of actions, variety of task constraints, and large state spaces of the MAS models in real cases, and for compressing large plans that could contain useless data. The compressed plans must have a correctness guarantee.

### 12.3.3 A Motivating Example

In this section, we introduce the *autonomous quarry* that serves as the industrial case-study provided by Volvo Construction Equipment (CE) in Sweden. As depicted in Fig. 12.3, the quarry contains various autonomous agents, e.g., trucks and wheel loaders. The goal of the agents is to transport stones from stone piles to crushers. Specifically, wheel loaders first dig stones at the stones piles and load them into trucks. Trucks can choose to get loaded from the wheel loaders or primary crushers. After being loaded, the trucks carry the stones to a secondary crusher, which is the destination of the stones.

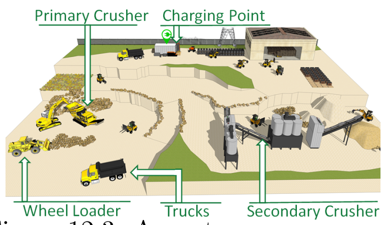


Figure 12.3: An autonomous quarry

During the transportation, the agents move, collaborate or work independently, and charge timely in order to achieve their goal, while satisfying requirements such as quarrying $2000m^3$ of stones per day. The challenges of the use case are as follows, which fall into the general challenges in planning problems of MAS (see Section 12.3.2):

- Task durations are uncertain because of the uncertainties in the environment. For instance, when trucks are unloading stones into a primary crusher, the speed of the conveyor belt on the primary crusher varies, which results in different execution times of unloading. Other trucks may need to wait until the previous one finishes its work at the primary crusher, which can even influence the entire plan (**Challenge I**).

- Some tasks are executed independently by agents, such as unloading to secondary crushers. Some tasks require collaboration between agents, such as wheel loaders loading stones into trucks. Some tasks must be prioritized when certain events occur, such as the charging task that must be prioritized when the agent's battery/fuel level is low (**Challenge II**).

- According to the experience of Volvo CE, the number of agents can vary from 2 to 8. However, our previous study has demonstrated that synthesizing correctness-guaranteed plans by using model checking is limited to MAS with less than 5 agents [9]. Handling larger numbers of agents is challenging (**Challenge III** and **Challenge IV**).

To overcome these challenges of MAS planning, we design an approach called MoCReL, which is an improved version of MCRL that we have proposed previously [12]. MCRL combines model checking with reinforcement learning, so it can deal with more agents than the algorithmic methods do, however, its task types do not support collaborations and events in MCRL, and large plans can not be compressed either. Next, we introduce MoCReL in detail.

## 12.4 Strategy Synthesis, Verification and Compression

In this section, we introduce the workflow of MoCReL and describe the TG of MAS together with the important techniques that are used in MoCReL for strategy synthesis, verification, and compression.

### 12.4.1 Overall Workflow of MoCReL

The workflow of MoCReL is shown in Fig. 12.4.
*Step 1*: A probabilistic quantification is conducted on the TG to facilitate sampling over the system, effectively turning the TG into an STG (Stochastic Timed Game).
*Step 2*: Strategy synthesis takes place, which employs the Monte-Carlo simulation in UPPAAL STRATEGO [14] to simulate the models and sample runs that satisfy certain properties. Next, the sampled runs are passed to the reinforcement learning module to generate strategies. Iterations between the simulation and learning continue until reaching the limit of iteration or sampling a user-defined number of runs. In this paper, we extend UPPAAL STRATEGO such that it supports using external libraries to change the learning module [25], and implement MoCReL as an external library[3]. *Step 3*: When the synthesis finishes, a stochastic strategy is obtained, which is then abstracted as a non-deterministic strategy and verified.

[3]The introduction and an example of the library are in Appendix A.4.

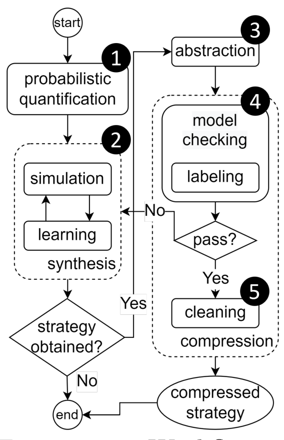


Figure 12.4: Workflow of MoCReL

*Step 4*: During the verification, the model checker inquires the external library, where the strategy is stored, about the allowed/preferred actions at a given state. The preferred state-action pairs are labeled as "visited".
*Step 5*: If the verification fails, we go back to the *Step 2* with an increased number of iteration limit so that the new round of synthesis can have more samples for learning. If the verification passes, the unlabeled pairs are removed from the strategy so that the compressed strategy takes less memory space.

Models and strategies throughout the workflow are interpreted semantically as shown later in Subsection 12.4.4. UPPAAL STRATEGO supports both the algorithmic synthesis in UPPAAL TIGA [26] and the learning-based synthesis that uses reinforcement learning [19]. Results of the algorithmic synthesis are correct-by-construction, but the method does not scale as it needs to explore the state spaces of the models exhaustively. In MoCReL, we propose a post-verification of the strategies that are synthesized by learning. The verification is exhaustive so the results are guaranteed to be correct. Moreover, as the verification is conducted on the agent model controlled by a strategy, the state space can be much reduced. Therefore, problems that are too complex to be handled by UPPAAL TIGA can be solved by MoCReL.

## 12.4.2 Modeling of MAS

MoCReL models the agent behaviors into timed games (TG), including: (i) movement TG that model the connection and traveling time between every pair of legal positions in the environment. Legal positions are the ones that are accessible by the agents; (ii) task execution TG that model the switch between tasks and the idle state, and the task execution time; (iii) monitor TG that monitor events. When an event occurs, a monitor TG informs task execution TG to execute the corresponding task.

As a major difference between MoCReL and our previous approach MCRL [12], the models in MoCReL are much easier to adapt to different scenarios of the planning problem, as they are instantiated from the model templates. One does not need to change the templates but only instantiate the templates with different values of parameters in order to fit in one's own application. The figures in this section only illustrate the brief structures of the model templates, without showing the complex guards and functions on the edges. The full templates are shown in Appendix A.3.

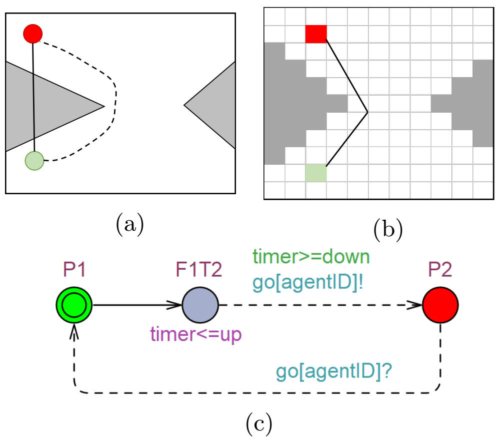


Figure 12.5: Examples of trajectories and the TG template of agent movement.

(i) **Movement TG**: The TG template of movement models agent traveling from one point to another. The points can be anywhere except the obstacles within the map. Since the purpose of the model is to synthesize plans, the movement TG do not model the concrete trajectories of the agents, but the traveling directions and times. Fig. 12.5c shows the brief structure of the template of *movement TG*, in which locations `P1` and `P2` represent any legal positions in the map. The location `F1T2` models the duration of traveling from `P1` to `P2`. Although the edge from `F1T2` to `P2` is uncontrollable by agents, the invariant (`timer` $\leq$ `up`) and guard (`timer` $\geq$ `down`) regulate that the traveling time must be within the interval between `down` and `up`. As one template only models one traveling direction, when the agent travels in the opposite direction, i.e., from `P2` to `P1`, the traveling time is counted by another TG that models the converse direction of movement. This TG (`P1` to `P2`) is synchronized with the other TG (`P2` to `P1`) on the channel `go` with the index of the agent.

Fig. 12.5 shows different modeling granularity of traveling from the green position to the red position. Even though there exists an obstacle between the two positions, one can model the movement as one instance of the movement TG template, which only reflects the existence of a movement between these two positions (the solid line in Fig. 12.5a), and the traveling time. In this case, solving the mission-planning problem aims at computing a high-level plan that does not concern how the agents maneuver in order to carry out the movement safely (the dashed line in Fig. 12.5a). Alternatively, one can model the movement as several instances of the template, which reflect the discrete segments of the trajectory (Fig. 12.5b). When using MoCReL to purely solve a path-finding problem, one can model the movement from one unit of the discrete map to another as an instance of the template. The granularity of modeling depends on the users' applications.

(ii) **Task execution TG**: Similar to the movement TG, the task execution TG do not model the concrete steps of executing a task, but only the switch between task execution and idle, and the execution time of the task. There are several different templates designed for different types of tasks, such as tasks without precondition, tasks with events, and tasks that need agents to collaborate. One can instantiate the templates according to one's own application by assigning values to the parameters of the templates, such as BCET and WCET (best-case and worst-case execution time, respectively), preconditions, and the event that activates the task. However, the structure of the templates is the same (Fig. 12.6).

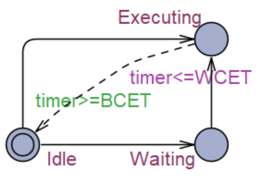


Figure 12.6: TG template of task execution

When the agent is allowed to execute a task, the edge from location `Idle` to `Executing` is enabled. A task being allowed means the following conditions are *true*: the task has not been finished yet, its precondition such as some certain tasks having been finished is *true*, the agent is at the right position where the task is allowed to run, and the events that activate the special tasks have not occurred (respectively, occurred) if the current one is a regular (respectively, special) task. Additionally, if a task needs a collaboration among agents, the collaborating agents must be ready and located at the same position.

When the task is ready but the device that is required by this task is taken by another agent, the agent can choose to wait, i.e., transfer to location `Waiting`, and change to location `Executing` when the device is free. When the task is being executed, the TG *can* leave location `Executing` when the timer exceeds the BCET, and *must* leave the location when it exceeds the WCET, meaning that the execution time of the task is between BCET and WCET. One thing that is worth mentioning is that agents are allowed to move only when not executing any task, i.e., the task execution TG is at the *Idle* location. We use a global Boolean variable shared by the movement TG and the task execution TG to indicate whether the agent is idle or not.

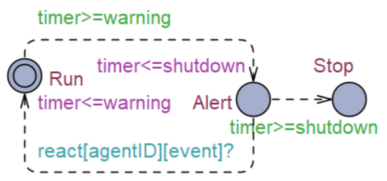


Figure 12.7: TG template of monitors

(iii) **Monitor TG**: A monitor monitors a signal and triggers an event when the signal exceeds a certain threshold. For simplicity, we assume the signals to be changing monotonically with time. Since the tool that MoCReL relies on, i.e., UPPAAL STRATEGO, allows defining ordinary differential equations (ODE) of continuous variables, one can eliminate this assumption by assigning ODE to locations. However, we leave this for the future work.

Based on the assumption, a monitor TG watches the elapse of time instead of the signal, and triggers an event when time elapses a certain period, meaning that the signal exceeds a certain threshold. In Fig. 12.7, when the `timer` exceeds a particular constant integer (i.e., `warning`), the monitor TG transfers to location `Alert` while updating a variable representing the event. The corresponding task execution TG (Fig. 12.6) is then activated in the sense that its edge for starting the task is enabled. If the agent can finish the task before the `timer` reaches the limit, which is represented by a constant integer (i.e., `shutdown`), the monitor TG moves back to the initial location to restart the monitoring; otherwise, the monitor goes to location `Stop`, when all controllable actions of the agent are not allowed to be taken any more, meaning that the agent stops operating.

We call the network of movement TG, task execution TG, and monitor TG a MAS TG. Properties of a MAS TG can be expressed by a subset of Computation Tree Logic (CTL) [15] that is supported by UPPAAL STRATEGO. Since the formal models of MAS have been defined, we can now define the planning problem formally before introducing the approach in detail.

**Definition 15** (Planning)**.** *Given a MAS TG $\mathcal{G}$ and a liveness property $\mathcal{Q}$ in the form of $A<>\ p$, the planning problem $\mathcal{M} =< \mathcal{G}, \mathcal{Q} >$ reduces to generate a strategy $\sigma$ over $\mathcal{G}$ such that $\mathcal{G}$ can satisfy $\mathcal{Q}$ when it is controlled by $\sigma$, i.e., $\mathcal{G} \mid \sigma \models \mathcal{Q}$.* □

The liveness property $A<>\ p$ means that $\mathcal{G} \mid \sigma$ will always eventually satisfy $p$. As the main goal of mission planning is to find the strategy that controls the agents to finish all their tasks eventually no matter how the environment reacts, the liveness property is used in the synthesis. The correctness guarantee of other requirements, such as safety, can be achieved by the verification after a plan is synthesized. We will give more details on these properties in Section 12.4.4.

### 12.4.3 Partial State-Space Observation

During the learning iteration, numerical rewards of taking an action at a state are used by reinforcement learning (e.g., the Bellman equation in Q learning [22]) to populate a score table of state-action pairs. When the learning finishes, the final values of the pairs are stored in the score table, which serves as a strategy. Before introducing how the strategy is used, in this subsection, we introduce another important concept in MoCReL: *partial observation* of the state space.

The learning algorithms need to identify the states of MAS to build up the score tables. As a formal model, MAS TG provides a clear definition of states, consisting of locations, clock values, and other data variables (Section 12.2). However, the strategies of MAS TG do not necessarily need to know all the components of states. For example, if the planning problem does not contain timing properties, e.g., finishing the tasks within 1 hour, the strategies can ignore all the clocks of the states, which simplifies the problem by eliminating unnecessary details. Hence, we use a partial observation of the state space of a MAS TG, which is supported by UPPAAL STRATEGO. One can simply provide the interesting variables of the MAS TG to the learning algorithm so that the synthesized strategies do not contain unnecessary information. Details of specifying partial observability is given in Query (12.3) in Subsection 12.4.4.

### 12.4.4 Key Techniques of MoCReL

In this section, we will give a detailed introduction of the key techniques used in MoCReL after the definition of strategies that we synthesize in this paper.

#### Strategy Definition

What MoCReL aims to synthesize is a subset of *memoryless* and *non-lazy* strategies that do not contain clocks. This restriction enables us to develop an algorithm for exhaustively verifying TG under the control of learned strategies in UPPAAL STRATEGO, which used to support exhaustive verification only on strategies with symbolic states. The trade-off of the restriction is that the planning problem in this paper does not consider timing properties. Formally, we define the strategies that MoCReL synthesizes as follows:

**Definition 16** ((Stochastic) Strategy with a Score Table). *Given $\mathcal{M} =< \mathcal{G}, \mathcal{Q} >$ as a planning problem of MAS, a (stochastic) strategy of $\mathcal{G}$ is a function $\sigma : q \to \mathcal{A} \subseteq \mathcal{A}^q_{\mathcal{G}}$ with a score table of state-action pairs, where $q$ is a state consisting of discrete variables, and $\mathcal{A}^q_{\mathcal{G}} \subseteq 2^{\Sigma_c \times \{\lambda\}}$ is a set of actions that are allowed by $\mathcal{G}$ at state $q$. The strategy $\sigma$ is considered to solve $\mathcal{M}$ if the following conditions hold, where $\|\mathcal{A}\|$ is the cardinality of $\mathcal{A}$, $max(\mathcal{A}^q_{\mathcal{G}})$ returns the actions with the highest score in $\mathcal{A}^q_{\mathcal{G}}$:*

1. *if $\|\mathcal{A}\| = 0$ (i.e., $\sigma$ does not contain $q$), then $\forall a \in \mathcal{A}^q_{\mathcal{G}}$, $a \in max(\mathcal{A}^q_{\mathcal{G}})$;*

2. *if $\|\mathcal{A}\| \geq 1$, then $\forall a \in \mathcal{A}$, $a \in max(\mathcal{A}^q_{\mathcal{G}})$;*

*When $\|\mathcal{A}\| \neq 1$, ties among actions happen. Non-deterministic (respectively, stochastic) strategies break the ties by non-deterministic (respectively, uniformly-distributed) choices over $\mathcal{A}$ when $\|\mathcal{A}\| > 1$, or $\mathcal{A}^q_{\mathcal{G}}$ when $\|\mathcal{A}\| = 0$.* □

Unlike the strategies synthesized by algorithmic methods (e.g., UPPAAL TIGA), the ones defined in Definition 16 do not guarantee to solve the MAS planning problem. Possible errors can exist in the design of the reward functions of the reinforcement learning algorithm, which do not reflect the desired properties in the planning problem, or the learning phase is not sufficient to populate a score table that covers enough states.

We will give some examples of the design errors in Section 12.4.4 after the query for synthesis is introduced.

In the next section, we show how MoCReL synthesizes, verifies, and compresses strategies defined in Definition 16.

## Probabilistic Quantification and Abstraction

Due to the inherent difference between the phases of synthesis and verification, models are interpreted semantically differently in MoCReL when being simulated from when they are being verified. This is automatically done by *probabilistic quantification* and *abstraction* of the models and strategies in MoCReL.

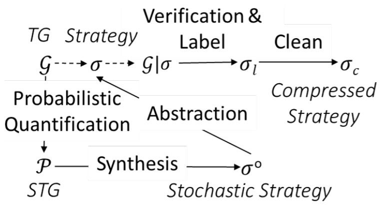


Figure 12.8: Model relations

Fig. 12.8 shows the transformation of the model semantics in the workflow of MoCReL. Initially, the MAS TG is interpreted as an STG during strategy synthesis because random simulation is needed in this step. An operation called *probabilistic quantification* changes the non-deterministic choices of actions to stochastic ones with concrete probability distributions. Specifically, time-bounded delays and discrete actions are transformed into stochastic ones with uniform distributions of probabilities. For example, in Fig. 12.6, the non-deterministic choice of when to leave location `Executing` is transformed to an uniformly-distributed one.

Exponential probability distributions are assigned to unbounded delays on locations with only uncontrollable actions as its outgoing edges, such as location `Executing` in Fig. 12.9. A constant integer of the exponential rate must be assigned to such locations (e.g., `1`). If a location has no invariant and only controllable actions as outgoing edges, such as location `Idle` in Fig. 12.9, the length of delay is not considered because we focus on non-lazy strategies, where agents urgently decide on a discrete action when it is available, or delay until the environment reacts. In this case, the discrete actions and delay are equally likely to be chosen initially, so the model templates do not need to be changed either.

In this paper, the duration of movement and task execution are all time bounded, so no exponential distribution exists in our MAS STG. Hence, the model templates of movement and task execution do not need to be changed, as the probabilistic quantification is done on the semantic level automatically by UPPAAL STRATEGO.

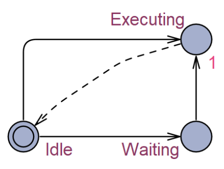


Figure 12.9: A task execution STG of a task with an unbounded execution time. This model does not exist in our MAS STG.

Next, synthesis based on the MAS STG generates stochastic strategies. Specifically, the simulation samples runs of the model and sends them to the learning algorithm to accumulate the scores of state-action pairs of the runs. During the learning phase, the probabilities of actions are not always the same. Actions with higher scores become more likely to be chosen than the ones with lower scores. Unexplored state-action pairs are equally likely to be chosen as the ones with the highest scores. This arrangement is referred to as "exploration" in reinforcement learning literature. After a user-defined number of runs is consumed by the learning algorithm, a stochastic strategy is considered to be generated.

After the synthesis, strategies are to be verified and compressed. To achieve verification, stochastic strategies must be transformed into non-deterministic strategies so that they can be exhaustively model checked. This step is called *abstraction* (see Fig. 12.8), which is also automatically carried out by UPPAAL STRATEGO on the semantic level. Abstraction eliminates the probabilistic information from a stochastic strategy by replacing the stochastic choices of actions with non-deterministic ones, and produces a strategy. Specifically, as defined in Definition 16, in the phase of verification, both strategies and stochastic ones always choose the actions with the highest scores. This is the so-called "exploitation" in reinforcement learning literature. When ties among actions appear, stochastic strategies equally likely choose one of these actions, whereas strategies make the decision non-deterministically. Therefore, a strategy may exhibit more behaviors than the stochastic strategy that the former is abstracted from. We prove this formally as follows:

**Theorem 1.** *Given a TG $\mathcal{G}$, an STG $\mathcal{P}$ obtained from $\mathcal{G}$ by the probabilistic quantification, a stochastic strategy $\sigma^\circ$ (Definition 16) solving $\mathcal{P}$, and a strategy $\sigma$ abstracting $\sigma^\circ$, the following inclusion holds: $Out(\mathcal{P} \mid \sigma^\circ) \subseteq Out(\mathcal{G} \mid \sigma)$.*

*Proof.* First, since $\mathcal{P}$ is obtained from $\mathcal{G}$ by the probabilistic quantification, an uncontrollable action that is chosen non-deterministically by $\mathcal{G}$ is chosen with equal probability by $\mathcal{P}$. If $\pi \in Out(\mathcal{P} \mid \sigma^\circ)$ and $q = last(\pi)$, there must be a $\pi' \in Out(\mathcal{P} \mid \sigma^\circ)$ such that $\pi = last(\pi') \xrightarrow{e} q$, where $e$ meets one of the three conditions in Definition 14. Assuming $\pi' \in Out(\mathcal{G} \mid \sigma)$, then

1. if $e \in \Sigma_u$, then $last(\pi') \xrightarrow{e} q \in Out(\mathcal{G} \mid \sigma)$ because $\sigma$ has no control on $e$, and $\mathcal{G}$ non-deterministically chooses $e \in \Sigma_u$. Hence, $\pi \in Out(\mathcal{G} \mid \sigma)$;
2. if $e \in \Sigma_c \cap \sigma(q)$ or $e = \lambda$, then according to Definition 16, $e$ has the highest score in $\mathcal{A}^q_{\mathcal{G}}$. Then $e$ can be chosen by $\sigma$ deterministically when $\|\mathcal{A}\| = 1$ or non-deterministically when $\|\mathcal{A}\| \neq 1$. Hence, $\pi \in Out(\mathcal{G} \mid \sigma)$.

Hence, $\pi = last(\pi') \xrightarrow{e} q \in Out(\mathcal{G} \mid \sigma)$. Likewise, we can inductively prove the assumption: $\pi' \in Out(\mathcal{G} \mid \sigma)$. Hence, if $\pi \in Out(\mathcal{P} \mid \sigma^\circ)$, $\pi \in Out(\mathcal{G} \mid \sigma)$, that is, $Out(\mathcal{P} \mid \sigma^\circ) \subseteq Out(\mathcal{G} \mid \sigma)$. $\square$

Theorem 1 shows that $\sigma$, as the abstraction of $\sigma^\circ$, may broaden $\sigma^\circ$'s outcome, since the former may exhibit behaviors that do not exist in the latter.

*Example.* Assume $\mathcal{P} \mid \sigma^\circ$ contains a state $q_i$ having 10 controllable actions, among which 4 of them have the highest score in $\sigma^\circ$, namely actions $a_i^1$, $a_i^2$, $a_i^3$, and $a_i^4$, respectively. Then a run as follows is possible in both $\mathcal{P} \mid \sigma^\circ$ and $\mathcal{G} \mid \sigma$,

$$\dots q_i \xrightarrow{a_i^1} q_{i+1} \to \dots \to q_i \xrightarrow{a_i^2} q'_{i+1} \to \dots \to q_i \xrightarrow{a_i^3} q''_{i+1} \to \dots \to q_i \xrightarrow{a_i^4} q'''_{i+1} \cdots$$

where state $q_i$ is reached 4 times when each of $a_i^1$, $a_i^2$, $a_i^3$, and $a_i^4$ gets to be chosen once. However, a run as follows is possible in $\mathcal{G} \mid \sigma$ but not in $\mathcal{P} \mid \sigma^\circ$:

$$\dots q_i \xrightarrow{a_i^2} q'_{i+1} \to \dots \to q_i \xrightarrow{a_i^2} q'_{i+1} \to \dots \to q_i \xrightarrow{a_i^2} q'_{i+1} \to \dots \to q_i \xrightarrow{a_i^2} q'_{i+1} \cdots$$

where $a_i^2$ is chosen 4 times in row. Therefore, the post-verification on $\sigma$ is necessary for ensuring that the resulting strategy meets the requirements. In the next section, we enumerate other reasons for conducting the post-verification.

### Strategy Synthesis

Synthesis in UPPAAL STRATEGO is done via the following queries:

```
strategy policy = minE(x)[<=T]{dv}->{cv}:<> P
```
(12.3)

```
strategy policy = maxE(x)[<=T]{dv}->{cv}:<> P
```
(12.4)

The keyword `minE(x)` (respectively, `maxE(x)`) means simulating the model while running the learning algorithm with the purpose of minimizing (respectively, maximizing) "`x`", which can be a variable or an expression. This is the so-called "reward function" in reinforcement learning literature. In addition, `T` is the maximum time for one round simulation, `dv` is a set of expressions regarded as discrete values, and `cv` is a set of expressions regarded as continuous values. These constitute the so-called "features" in reinforcement learning literature [21].

The state space of the MAS TG is partially shown to the learning algorithm by the values of the expressions in `dv` and `cv`. In particular, MoCReL only allows discrete variables, hence the synthesized strategies do not contain clocks. This limitation facilitates the verification of the learned strategy since the preference of choice of controllable action cannot change within zones that represent the basic construction enabling symbolic verification of timed automata [15].

The formula "`<> P`" is a CTL property, and only the runs that satisfy this property are sampled in the simulation. These runs are used as input of the learning algorithm to calculate the scores of state-action pairs. In particular, MoCReL uses "`<> time` $\geq$ `C`", where `time` is a global clock in the model that is never reset and `C` $\in$ [0, `T`] is a constant integer within the simulation time `T`. This formula allows all runs that simulate the model over `C` time units to be passed to the learning algorithm no matter whether the agents reach their goals or not. Hence, both good and bad state-action pairs are passed to the learning algorithm, which accumulates their scores by using their rewards or penalties, respectively.

When running Query (12.3) in UPPAAL STRATEGO, our new version of the tool calls an external library, which implements the learning algorithm of MoCReL to synthesize strategies, and stores the score table of the strategy. With the help of the external library, one can plug in one's own learning algorithm or add new functions into the existing algorithm. We show this in Section 34.

*Example.* Now, we revisit the path-finding problem of Section 12.3.2, Fig. 12.2, to shown on the example concretely the necessity of verifying the resulting strategies, which in fact follows from the one way inclusion of Theorem 1. Assume that the cat stays at its current position for `N` minutes, and that the robot wants to catch it as quickly as possible, then the reward function can be specified as:

$$\texttt{x} = \texttt{time} - \texttt{caught} \times \texttt{REWARD} \tag{12.5}$$

The variable `time` is the global clock aforementioned, `caught` is a binary integer (i.e, 1/0) indicating if the cat is caught by the robot or not, and `REWARD` is a non-negative integer that the robot gets when it catches the cat. It is trivial to see that the smaller the value of `x` is, the better the strategy is.

If one mistakenly adopts the reward function of equation 12.5 but Query (12.4) for synthesis, which attempts to find the state-action pairs maximizing `x`, the result can still be obtained, as the synthesis is only about accumulating scores of the pairs and populating a score table. However, the actions that consume the longest time (i.e., `time` being maximum) but never catch the cat (i.e., `caught` being 0) are taken as the best actions in this result.

This example shows a possible misuse of the queries for synthesis. Even if one uses the query and reward function correctly, the resulting mission plan may still be wrong, because the samples for learning may be too few to populate a score table that covers enough states, or the MAS model is wrongly designed and violates other requirements of the agents that are not reflected in the queries for synthesis. In a nutshell, the learning-based synthesis does not have a correctness-guarantee on its results.

### Strategy Verification

Different from MCRL [12], the verification in MoCReL is directly conducted on the MAS model under the control of a strategy, because UPPAAL STRATEGO supports the following verification queries [14]:

$$\texttt{A<>}\ \phi\ \texttt{under}\ \sigma \tag{12.6}$$

$$\texttt{Pr[<=T]}\ \phi\ \texttt{under}\ \sigma \tag{12.7}$$

The keyword `under` puts the state space exploration of the MAS TG under the control of the strategy that is synthesized and stored by the external library of MoCReL. Query (12.6) returns an absolute answer of true or false to the question of whether $\phi$ is always eventually satisfied, whereas Query (12.7) returns the probability of satisfying $\phi$.

In this paper, we extend UPPAAL STRATEGO to support Query (12.6) on strategies that are synthesized by learning. The pseudo-code of executing Query (12.6) is in Algorithm 6. In Appendix A.2, we illustrate the execution of the algorithm with an example. Here, we overview the algorithm briefly. To verify a liveness property like Query (12.6), one needs to explore the model's state space until either getting a counter-example run violating the property, or until reaching all the states. Specifically, a counter-example of a liveness property like Query (12.6) must be either a loop, or a maximum run ending at an unbounded state or a deadlock, in which all the states do not satisfy $\phi$. Hence, once such a run is found, the verification terminates with a negative result (line 15 and line 17).

Additionally, the state space exploration must be guided by a strategy. When the model checker faces controllable actions (i.e., in line 22, *isControllable*($\xrightarrow{a}$) returns *true*)), or a delay (lines 7 and 9), it calls a function `Allow` to lookup the score table of a strategy. According to Definition 16, actions with the highest scores are always chosen – with ties broken by a uniform distribution (Query (12.7)) or a non-deterministic choice (Query (12.6)). In this way, the liveness verification is guided by a strategy.

*Example.* We show several queries that can be used in the verification of the synthesized strategy in the path-finding problem of Section 12.3.2, Fig. 12.2.

$$\begin{array}{r}\texttt{strategy policy = minE(time - caught} \times \texttt{REWARD)[<=100]} \\ \texttt{\{robot.location\}->\{\}:<> time} \geq \texttt{90}\end{array} \tag{12.8}$$

$$\texttt{A<> caught under policy} \tag{12.9}$$

$$\texttt{A[] !collide() under policy} \tag{12.10}$$

Query (12.8) synthesizes a strategy named `policy`, which is supposed to catch the cat within 100 time units. Query (12.9) verifies the robot

**Algorithm 6:** Algorithm of liveness verification (adapted from Fig. 3 in the literature [27]): model checking $\mathcal{G} \mid \sigma$ against Query (12.6)

```
Function Liveness($\mathcal{G}$, $\sigma$, $\phi$):
    $ST := \emptyset$ $SD := \emptyset$ $Passed := \emptyset$
    Delay($\mathcal{G}.S_0, \neg\phi$)
    for $S_d \in SD$ do
        Search($S_d, \neg\phi$)
    return ($true$)
Function Delay($S, \varphi$):
    for $S' : S \xrightarrow{d} S'$ do
        if Allow($\sigma, \xrightarrow{d}$) then
            if $(S' \notin SD) \wedge (S' \models I(S.l) \wedge \varphi)$ then
                $push(SD, S')$
Function Search($S, \varphi$):
    $S := S \wedge \varphi$
    if $S \neq empty$ then
        if $loop(S,ST)$ then
            exit($false$)                                  // Loop found
        if $unbounded(S) \vee deadlocked(S)$ then
            exit($false$)                          // Maximal run found
        $push(ST, S)$
        if $\forall S' \in Passed : S \nsubseteq S'$ then
            for $S_a : S \xrightarrow{a} S_a$ do
                    // If action a is uncontrollable or allowed, it can be
                chosen.
                if $\neg isControllable(\xrightarrow{a}) \vee$ Allow($\sigma, \xrightarrow{a}$) then
                    Delay($S_a$, $\varphi$)
                    for $S_d \in SD$ do
                        Search($S_d$, $\varphi$)                // Recursive all
    $Passed := Passed \cap \{pop(ST)\}$              // Move from stack to Passed
Function Allow($S, action$):
    if $NumControllable(S) == 1$ then
        return ($true$)
    if $action \in best(\sigma, S)$ then
        $label(\sigma, S, action)$                     // Label (S, action) in $\sigma$
        return ($true$)
    else
        return (false)
```

model under the control of `policy` to see if it can always eventually catch the cat. Query (12.10) involves a function `collide()` implemented in the model, which detects the distances from the robot to obstacles in the environment and returns *true* if any one of the distances is less than a certain value, or *false* otherwise. This query verifies that the collision between the robot and obstacles never happens.

Besides the possible errors in the resulting strategies, as presented in the path-finding example, strategies can be memory consuming for containing too many useless data. With the help of the external library where MoCReL is implemented, we can leverage queries in the form of Query (12.6) to not only verify the strategy but also compress the strategy.

### Strategy Compression

Once an external library is linked to UPPAAL STRATEGO, the model checker can enquire the external library when facing multiple controllable actions. For example, when more than one agent is ready to execute a task, the model checker without an external library simply traverses all options non-deterministically, whereas the model checker with an external library passes the current state and the available actions of the state to the external library one by one, and obtains the preference of each state-action pair. The ones with the highest score are always preferred. In MoCReL, besides returning the preference, we also label the state-action pairs that have the highest score as “selected” because they will be selected and verified by the model checker.

When verifying a liveness property (e.g., Query (12.6)), the model checker must explore all the branches of the state space to ensure that the proposition of the property (e.g., $\phi$ in Query (12.6)) is always eventually *true*. Therefore, if the liveness property is satisfied, the labelled state-action pairs are “selected” from the state space and the exhaustiveness of search guarantees them to always eventually reach the states where the property is true. The unlabelled pairs are considered “useless” data because without them, the property can still be satisfied. Therefore, the strategy can be compressed by removing the unlabelled pairs (*cleaning* in Fig. 12.4). By verifying the compressed strategy again, we can see that the new strategy preserves the liveness property that is met by the original strategy.

**Algorithm 7:** MoCReL algorithm

```
Function Main($\mathcal{G}$, $\mathcal{Q}$, $iterationNum$, $totalNum$, $goodNum$, $formula$):
    Strategy $\sigma := \emptyset$, $\sigma_c := \emptyset$
    Stochastic Strategy $\sigma^\circ := \emptyset$
    STG $\mathcal{P} := ProbabilisticQuantification(\mathcal{G})$
    while $\neg$Liveness($\mathcal{G}$, $\sigma$, $\mathcal{Q}$) do
        $\sigma^\circ := Learn(\mathcal{P}, iterationNum, totalNum, goodNum, formula)$
        $\sigma := Abstraction(\sigma^\circ)$
        $Update(iterationNum, totalNum, goodNum)$
    $\sigma_c := Clean(\sigma)$
    return ($\sigma_c$)
```

## Soundness of MoCReL

Algorithm 7 is the pseudo-code of MoCReL. Line 4 and line 7 are the probabilistic quantification and abstraction, respectively. Line 6 runs an algorithm that iteratively simulates and learns until a user-defined number of samples are obtained, or the iteration reaches its maximum rounds (see Algorithm 8 in Appendix A.1). The function `Liveness(`$\mathcal{G}, \sigma, \mathcal{Q}$`)` at line 5 runs Algorithm 6, which verifies if $\mathcal{G} \mid \sigma \models \mathcal{Q}$ as defined in Definition 15, and labels the state-action pairs that are selected by the model checker. Line 8 updates the parameters for learning, e.g., increasing the number of samples (i.e., `totalNum`) to have a larger score table that covers more states than that of the last strategy. Line 9 compresses $\sigma$ by removing the unlabeled data.

**Soundness of the Approach**. When MoCReL terminates with a synthesized strategy, the result is verified, which guarantees that the planning problem (Definition 15) is answered correctly. Formally, MoCReL is sound, proven by Theorem 2 below:

**Theorem 2** (Soundness)**.** *Given a planning problem $Q =< \mathcal{G}, \ \mathcal{Q} >$, where $\mathcal{Q} = A <> \phi$, if Algorithm 7 terminates and returns a strategy $\sigma_c$, then $\mathcal{G} \mid \sigma_c \models \mathcal{Q}$.*

*Proof.* Obviously, Algorithm 7 terminates with two cases:

1. `Liveness(`$\mathcal{G}$, $\sigma$, $\mathcal{Q}$`)` returns *true* (line 5 in Algorithm 7), when Algorithm 7 will eventually return $\sigma_c$ (line 10 in Algorithm 7);

2. `Liveness(`$\mathcal{G}$, $\sigma$, $\mathcal{Q}$`)` exits with a negative result (line 16 and line 18 in Algorithm 6), and no strategy is returned (line 10 in Algorithm 7 never being reached).

In Case 2, no strategy is generated, hence, we only need to prove when Case 1 happens, $\mathcal{G} \mid \sigma_c \models A <> \phi$. Assuming `Liveness` returns *true*, but $\mathcal{G} \mid \sigma_c \not\models A <> \phi$, then $\mathcal{G} \mid \sigma_c \models E[]\neg\phi$, which holds if and only if the following two conditions hold (the code lines referred to in the rest of the proof are all of Algorithm 6):

(i) The labeling is complete, that is, all the controllable state-action pairs that are selected by the model checker are labeled, but $\mathcal{G} \mid \sigma_c \models E[]\neg\phi$, which reads that there exists a run in $\mathcal{G} \mid \sigma_c$, in which all the states do not satisfy $\phi$;

(ii) The labeling is incomplete, that is, some pairs that are selected by the model checker are not labeled, which makes the model checker use the wrong actions at certain states when verifying $\mathcal{G} \mid \sigma_c \models A <> \phi$ and get a negative result.

In Case (i), such a run is either a loop or a run ending in a deadlock or an unbounded state, in which all the states do not satisfy $\phi$. Then the `Search` function must exit with a verification result of false (line 16 and line 18), which contradicts that `Liveness` returns true assumption.

In Case (ii), wherever the model checker faces a controllable action (line 22) or a delay (lines 7 and 9), it invokes the function `Allow`, which returns true when the state has only one controllable action (line 28), or the action is labeled (line 31). Hence, when facing multiple controllable actions, the model checker can never select an unlabeled action. Therefore, Case (ii) cannot happen.

In a nutshell, Case (i) and Case (ii) cannot happen, and thus $\mathcal{G} \mid \sigma_c \models E[]\neg\phi$ does not hold, that is, $\mathcal{G} \mid \sigma_c \models A <> \phi$ must hold when the function `Liveness` returns true, that is, when Algorithm 7 terminates and returns $\sigma_c$. $\square$

## 12.5 Experimental Evaluation

In this section, we evaluate MoCReL in several experiments to see its performance in the use case of an autonomous quarry with different numbers of agents, tasks, and task execution time. The reinforcement learning algorithm used in the experiments is Q-learning [22]. The experiments are conducted on an Intel Xeon E5-2678 with 256 GB of RAM running Ubuntu 20.04 LTS. All the models, tool, and the full experiment results can be found at: https://github.com/rgu01/MoCReL-Experiments.git.

### 12.5.1 Use Case Description

Fig. 12.10 depicts an autonomous quarry that is abstracted from a real scenario, where there are two kinds of autonomous agents: wheel loaders and trucks. Wheel loaders dig stones and load them into trucks. The latter load stones either from the wheel loaders or from a primary crusher, before transporting the stones to their destination: a secondary crusher. The goal of the agents is to transport a certain amount of stones. Agents need to go to a charging station for refueling when the energy level is low.

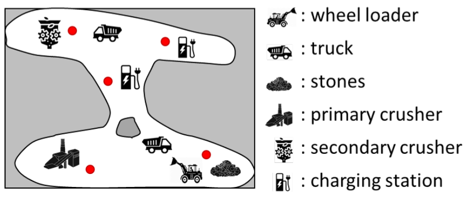


Figure 12.10: An autonomous quarry

To solve the MAS planning problem in this use case, first we model the system in the way described in Section 12.4.2. For simplicity, the sub-problem of path finding is solved by the A* algorithm [3] and our movement TG only models the traveling between every pair of milestones where tasks are carried out (e.g., red dots in Fig. 12.10). Although simplified, the state space of the problem during verification still grows exponentially with the linear increase of the number of agents [9]. Task execution TG models four types of tasks: (i) individual tasks with no precondition, e.g., wheel loaders digging stones; (ii) individual tasks with preconditions, e.g., trucks unloading stones into the secondary crushers with a precondition: the unloading task can be carried out only after the trucks have been loaded by wheel loaders or at primary crushers; (iii) collaborating tasks, e.g., wheel loaders loading stones into trucks; (iv) tasks that are activated by events, e.g, refueling when an agent's energy level is low. In addition, we design a special TG named *Referee* (Fig. A.5 in Appendix A.3), which judges if the goal is reached (i.e., enough stones are transported) or the maximum simulation time has been reached. In either case, the agents must stop, i.e., no controllable actions can be taken. The learning algorithm partially observes the state space of the models by detecting discrete variables such as the locations of the TG[4].

According to our previous study, the method that purely uses search-based algorithms can only solve a simplified version of this problem,

[4]Discrete variables can be seen in the queries of the models in the artifacts.

where task execution time is fixed and the number of agents is less than 5 [9, 13]. MCRL [12] can deal with more agents and flexible task execution time, but collaborations and events are not supported. These experiments include the collaboration among agents and a battery-low event. Maps in the experiments are also complex, i.e., some models contain 2-4 primary crushers and 1-2 secondary crushers.

### 12.5.2 Experiment Design

We conduct two series of experiments: 1) one where we study the synthesis time and compression efficiency, and 2) one where we study the influence of the number of sampled runs on the learning efficiency. Models that are used in both series of experiments are generated automatically by randomly assigning values to the parameters of the environment, e.g., the number of agents. The parameters are reported in Table 12.1 that we introduce in the next sub-section. The abbreviations in Table 12.1 are given as a footnote[5].

The first series of experiments is conducted on the full set of models while the second is restricted to a subset. The set of models is grouped intro three categories:

- *Category I*: Models with large numbers of agents up to 6, a small number of crushers (2), a fixed medium value of the trucks' capabilities (20), and no *monitor* TG for charging.
- *Category II*: Models with medium numbers of agents (2 - 5), large numbers of crushers (3 - 6), a range of the trucks' capabilities (10 - 30), and no *monitor* TG for charging.
- *Category III*: Models with small numbers of agents (2 - 3) and crushers (2), a fixed large value of the trucks' capabilities (50), and 1 - 2 *monitors* TG for charging.

The second series of experiments is conducted on a model `game6-B` in Table12.1 and its variants that change the amount of stones trucks can

[5]Abbreviations in Table 12.1: category (CAT), the number of wheel loaders (WL), the number of trucks (TK), the number of primary crushers (PC), the number of secondary crushers (SC), the number of chargers (CH), the capability of trucks (CAP), if the task execution time is time intervals or not (INT), the number of runs (RUNS), the computation time of synthesis in seconds (STIME), the size of the original strategy in MB (ORI), the size of the compressed strategy in MB (COM), the result of verification (VER).

Table 12.1: Results of strategy synthesis, verification, and compression

| CAT | model | WL | TK | PC | SC | CH | CAP | INT | RUNS | STIME | ORI | COM | VER |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I | game1-A | 2 | 4 | 1 | 1 | 0 | 20 | YES | 2000 | 3,902 | 27 | 0.13 | TRUE |
| | game3-A | 1 | 2 | 1 | 1 | 0 | 20 | YES | 200 | 16 | 0.08 | 0.02 | TRUE |
| | game4-A | 2 | 4 | 1 | 1 | 0 | 20 | YES | 5,000 | 772 | 5.6 | 0.03 | TRUE |
| | game6-A | 2 | 1 | 1 | 1 | 0 | 20 | YES | 200 | 175 | 0.09 | 0.02 | TRUE |
| | game7-A | 1 | 4 | 1 | 1 | 0 | 20 | YES | 5,000 | 575 | 4.7 | 0.03 | TRUE |
| | game8-A | 1 | 2 | 1 | 1 | 0 | 20 | YES | 200 | 14 | 0.08 | 0.02 | TRUE |
| | game9-A | 1 | 4 | 1 | 1 | 0 | 20 | YES | 5,000 | 640 | 4.4 | 0.05 | TRUE |
| II | game0-B | 1 | 2 | 3 | 1 | 0 | 10 | YES | 500 | 92 | 0.9 | 0.2 | TRUE |
| | game1-B | 1 | 1 | 4 | 1 | 0 | 10 | YES | 500 | 71 | 0.02 | 0.1 | TRUE |
| | game3-B | 1 | 2 | 1 | 2 | 0 | 10 | YES | 100,000 | 17,297 | 1.4 | 0.6 | TRUE |
| | game1-E | 1 | 3 | 1 | 2 | 0 | 30 | NO | 500 | 88 | 5.9 | 0.03 | TRUE |
| | game5-E | 1 | 3 | 4 | 2 | 0 | 30 | NO | 5000 | 1,705 | 103 | 0.05 | TRUE |
| | game2-B | 1 | 4 | 1 | 2 | 0 | 10 | YES | 100,000 | 800 | 112 | - | FALSE |
| | game6-B | 1 | 3 | 3 | 2 | 0 | 10 | YES | 100,000 | 893 | 121 | - | FALSE |
| III | game4-C | 1 | 2 | 1 | 1 | 2 | 50 | YES | 2,000 | 270 | 9.4 | 0.03 | TRUE |
| | game5-C | 1 | 2 | 1 | 1 | 1 | 50 | YES | 5000 | 410 | 2.8 | 0.03 | TRUE |
| | game3-D | 1 | 2 | 1 | 1 | 1 | 50 | NO | 500 | 68 | 1.4 | 0.03 | TRUE |
| | game6-D | 1 | 2 | 1 | 1 | 2 | 50 | NO | 500 | 80 | 2.6 | 0.03 | TRUE |
| | game9-D | 1 | 2 | 1 | 1 | 2 | 50 | NO | 500 | 84 | 7.0 | 0.03 | TRUE |
| | game6-C | 1 | 1 | 1 | 1 | 2 | 50 | YES | 100,000 | 8,629 | 0.7 | - | FALSE |
| | game8-C | 1 | 2 | 1 | 1 | 2 | 50 | YES | 100,000 | 12,457 | 49 | - | FALSE |

carry at one time. For these three models, we modify the "RUNS" from 100 to 500, and for each number of "RUNS", we synthesize a strategy and statistically verify its probability of reaching the goal by using queries in the form of Query (12.7). We repeat this experiment 10 times and use the mean values of the probabilities to be the result of verification to account for the random nature of statistical model checking.

### 12.5.3 Experiment Results

In Table 12.1, column "CAP" indicates the amount of stones that trucks can transport at one time, and the target amount of stones to be transported is the same in all models. Column "VER" shows the results of verifying queries in the form of Query (12.6). Column "RUNS" includes the numbers of runs that are needed to synthesize a strategy, which are picked empirically.

**Synthesis time**. In category I, the time of synthesizing strategies is relatively short, respectively. Most of the cases spend several seconds and the most difficult one (`game1-A`) costs more than 1 hour with the largest strategy (27M) produced in this category. In category II, synthesis time remains at the level of minutes for most of the cases. One interesting comparison is between `game3-B` and `game5-E` in this category. Considering the numbers of agents and milestones (e.g., crushers), the latter is more complex than the former. However, `game3-B` needs 100,000 runs and more than 4 hours to synthesize a successful strategy that passes the verification, whereas `game5-E` only needs 5000 runs and half an hour. The reason is because the task execution times are fixed in `game5-E` whereas the ones in `game3-B` are time intervals. The time intervals cause many interleaving actions which increase the state space of the model dramatically. When maps have chargers in category III, the synthesis times for successful strategies are at most several minutes. However, some models in categories II and III can be very complex so that learning with $100,000$ runs cannot generate successful strategies. We will discuss this in the presentation of learning efficiency.

**Verification results**. Overall, most of the cases ($\frac{41}{50}$) in the experiments pass the verification[6]. In some cases (e.g. `game2-B` in category II), we find counter-examples in the strategies that violate the liveness property, so they do not pass the verification. Increasing their simulation time and rounds to gather more runs for learning can be helpful in these cases. However, the fact that the models in these cases have large state spaces makes reaching the goal state a rare event that is hard to catch by random simulation (see the results of learning efficiency). This phenomenon stems from the nature of reinforcement learning algorithms that rely on random simulation.

**Learning efficiency**. Fig. 12.11 shows the mean probabilities of agents reaching their goal (i.e., satisfying Query (12.7)). The original model is `game6B`, in which the capability of trucks is 10, and the modified models are `game6B-7` and `game6B-8`, which decrease the capability to 7 and 8, respectively. The results of model `game6B` are not shown in the figure because all the experiments with 100 to 500 runs generate the same result: above 97%. The probabilities of `game6B-7` and `game6B-8` increase with the increasing numbers of runs. The probabilities of model `game6B-7` are lower than those of the other two models and the IQR (interquartile ranges[7]) are the largest. This indicates that when reaching the goal becomes hard, learning efficiency becomes unstable in the sense that the probabilities of satisfaction under the learned strategy vary dramatically.

One interesting observation is that, although the original model of `game6B` cannot generate a successful strategy not even when the number

[6]Full results of all models can be seen: shorturl.at/dkqyE

[7]IQR is the difference between the 75th and 25th percentiles of the data.

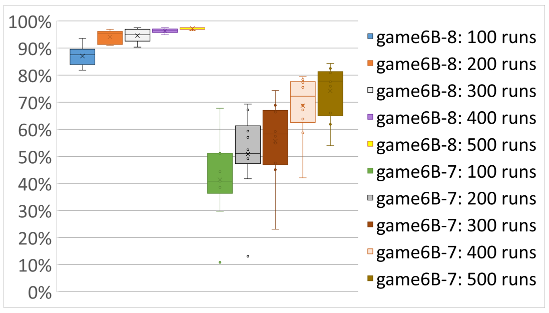


Figure 12.11: Distribution of mean probabilities of satisfaction over 10 experiments

of runs is 100,000, its mean probabilities of satisfaction for the strategies synthesized by a few runs (i.e., 100 - 500) are quite high (i.e., above 97%) with a standard deviation of 0. This phenomenon shows that when reaching the goal becomes a rare event, the benefit of increasing the number of runs becomes very low.

**Strategy compression**. The reduced sizes of compressed strategies are up to 99.95% of the original sizes in our experiments (e.g., `game5-E` in category II). Strategies that do not pass the verification are not compressed and thus are shown as "-" in the column "COM" of Table 12.1. The compressed strategies not only save memory space but also improve the explainability of the strategies. For example, the score table of the complete strategy in `game4-A` has almost 78,000 rows of data, which is reduced to less than 50 rows in the compressed strategy[8]. The latter is much more readable and explainable by humans.

**Conclusion of the Experiments**. The experiments show that MoCReL can solve the MAS planning problem in complex maps with multiple crushers and chargers. Successful strategies are verified and compressed and the reduced sizes are significant. Counter-examples of the liveness property can be found in unsuccessful strategies, which indicate where the agents fail. Compared to MCRL, although the environment is more complex, the task types are richer, and the number of agents is larger, MoCReL can still solve most of the cases in a reasonable time. The learning efficiency of reinforcement learning drops dramatically when reaching the goal state becomes a rare event in the

[8]Please see the printed strategies of `game4-A` in the artifacts.

model.

## 12.6 Related Work

Synthesis of strategies for MAS has been an increasingly researched area in recent years. Andersen *et al.* [28] present a UPPAAL-based method for motion planning of multi-robot systems. Their method uses reachability queries to generate motion plans, which is not sufficient for synthesizing comprehensive strategies that consider time intervals as the execution time of motions. Alur *et al.* [29] use the game theory for compositional synthesis of reactive controllers from Linear Temporal Logic (LTL) specifications for multi-agent systems, in which agents can be controllable or uncontrollable. Gleirscher *et al.* [30] introduce an approach for synthesis and verification of safety controllers for human-robot collaboration. Křetínskỳ [31] investigate the combination of LTL, Steady-State Policy Synthesis (SSPS), and long-run average reward (LRA) on synthesizing policies that resolve Markov decision processes (MDP). Bersani *et al.* [32] present the PuRSUE (Planner for RobotS in Uncontrollable Environments) approach, which supports users to configure their robotic applications and automatically generate their controllers by using UPPAAL TIGA. The main difference between our work and theirs is that their synthesis is based on search, which is correct-by-construction, but the scalability is limited.

In the field of combining formal methods with reinforcement learning (RL), Behjati *et al.* [33] attempt to solve the state-space-explosion problem of model checking LTL properties by using RL. Bouton *et al.* [34] propose a method that enforces probabilistic guarantees on agents during the course of RL. Jothimurugan *et al.* [35] propose *DIRL*, a synthesis approach that interleaves Djikstra's algorithm with RL to train agents. In comparison, the correctness guarantee provided by MoCReL is not on the course of learning or formal specification of the rewards functions and agent tasks. MoCReL provides post-verification of the synthesis results, which is more scalable than verifying the original agent models.

In the area of strategy compression, Julian *et al.* explore several ways of compressing strategies by using origami compression [36] or deep neural network [23][37]. Ashok *et al.* propose a decision-tree-based method for concisely representing strategies [38][39]. Their tool *dtControl* is able to compress strategies produced by UPPAAL TIGA. Compared with these methods, the strategy compression in MoCReL focuses on cleaning the unused data in the strategies rather than representing them in different forms. Compression in MoCReL replying on exhaustive model checking inherently provides safety guarantee of the strategies, which needs extra effort to achieve in other methods [37].

## 12.7 Conclusions and Future Work

We present a new method, namely MoCReL, for synthesis, verification, and compacting of strategies of multi-agent autonomous systems (MAS). MoCReL uses reinforcement learning for synthesizing strategies and model checking for verifying and compressing the strategies. MoCReL is integrated into UPPAAL STRATEGO, which facilitates the use of this method. Experiments carried out on a real-word autonomous quarry case study show that MoCReL is able to solve the planning problem of MAS in complex maps with large numbers of agents. The compressed strategies save up to 99.95% of the memory space taken by the original strategies. When reaching the goal state becomes a rare event that is hard to be captured by random simulation, the learning efficiency of reinforcement learning drops dramatically.

An interesting direction of the future work is to investigate the use of the counter-examples to repair the unsuccessful strategies, which would increase the learning efficiency profoundly. Introducing clocks into the strategies can be another challenging direction of research.

## Acknowledgments

We acknowledge the support of the Swedish Knowledge Foundation via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr: 20150022, and via the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190-038.

# Bibliography

[1] Eugenio Oliveira, Klaus Fischer, and Olga Stepankova. Multi-agent systems: which research for which applications. *Robotics and Autonomous Systems*, 27(1-2):91–106, 1999.

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

[4] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report*, 1998.

[5] Jean-Francois Kempf, Marius Bozga, and Oded Maler. As soon as probable: Optimal scheduling under stochastic uncertainty. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 385–400. Springer, 2013.

[6] Kim Guldstrand Larsen, Adrien Le Coënt, Marius Mikučionis, and Jakob Haahr Taankvist. Guaranteed control synthesis for continuous systems in uppaal tiga. In *Cyber Physical Systems. Model-Based Design*, pages 113–133. Springer, 2018.

[7] Wei Zhang and Thomas G Dietterich. High-performance job-shop scheduling with a timedelay td () network. *Advances in neural information processing systems*, 8:1024–1030, 1996.

[8] Chathurangi Shyalika, Thushari Silva, and Asoka Karunananda. Reinforcement learning in dynamic task scheduling: A review. *SN Computer Science*, 1(6):1–17, 2020.

[9] Rong Gu, Eduard Paul Enoiu, and Cristina Seceleanu. Tamaa: Uppaal-based mission planning for autonomous agents. In *35th ACM/SIGAPP Symposium On Applied Computing SAC2020*. ACM, 2019.

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

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

[12] Rong Gu, Eduard Paul Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Verifiable and scalable mission-plan synthesis for multiple autonomous agents. In *25th International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2020.

[13] Rong Gu, Peter Jensen, Danny Poulsen, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Verifiable strategy synthesis for multiple autonomous agents: A scalable approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 24(3), 2022.

[14] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal stratego. In *TACAS 2015: International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015.

[15] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 87–124, 2004.

[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] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR 2005: International Conference on Concurrency Theory*, pages 66–80. Springer, 2005.

[18] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[19] Alexandre David, Peter G Jensen, Kim Guldstrand Larsen, Axel Legay, Didier Lime, Mathias Grund Sørensen, and Jakob H Taankvist. On time with minimal expected cost! In *International Symposium on Automated Technology for Verification and Analysis*, pages 129–145. Springer, 2014.

[20] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

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

[22] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards.* King's College, Cambridge United Kingdom, 1989.

[23] Kyle D Julian, Mykel J Kochenderfer, and Michael P Owen. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.

[24] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[25] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. Teaching stratego to play ball: Optimal synthesis for continuous space mdps. In *International Symposium on Automated Technology for Verification and Analysis*, pages 81–97. Springer, 2019.

[26] Gerd Behrmann, Alexandre David, Emmanuel Fleury, Kim Larsen, Didier Lime, and Ecole Nantes. Uppaal-Tiga: Time for playing games! (tool paper). In *Proceedings of the 2007 Computer Aided Verification.* Springer Berlin Heidelberg, 2007.

[27] Gerd Behrmann, Kim G Larsen, and Jacob Illum Rasmussen. Beyond liveness: Efficient parameter synthesis for time bounded liveness. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 81–94. Springer, 2005.

[28] 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.

[29] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis of reactive controllers for multi-agent systems. In *International Conference on Computer Aided Verification*, pages 251–269. Springer, 2016.

[30] Mario Gleirscher, Radu Calinescu, James Douthwaite, Benjamin Lesage, Colin Paterson, Jonathan Aitken, Rob Alexander, and James Law. Verified synthesis of optimal safety controllers for human-robot collaboration. *arXiv preprint arXiv:2106.06604*, 2021.

[31] Jan Křetínskỳ. Ltl-constrained steady-state policy synthesis. *arXiv preprint arXiv:2105.14894*, 2021.

[32] Marcello M Bersani, Matteo Soldo, Claudio Menghi, Patrizio Pelliccione, and Matteo Rossi. Pursue-from specification of robotic environments to synthesis of controllers. *Formal Aspects of Computing*, 2020.

[33] 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.

[34] 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.

[35] Kishor Jothimurugan, Suguman Bansal, Osbert Bastani, and Rajeev Alur. Compositional reinforcement learning from logical specifications. *Advances in Neural Information Processing Systems*, 34, 2021.

[36] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2016.

[37] Kyle D Julian and Mykel J Kochenderfer. Guaranteeing safety for neural network-based aircraft collision avoidance systems. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10. IEEE, 2019.

[38] Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínskỳ, Maximilian Weininger, and Majid Zamani. dtcontrol: Decision tree learning algorithms for controller representation. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–7, 2020.

[39] Pranav Ashok, Mathias Jackermeier, Jan Křetínskỳ, Christoph Weinhuber, Maximilian Weininger, and Mayank Yadav. dtcontrol 2.0: Explainable strategy representation via decision tree learning steered by experts. *arXiv preprint arXiv:2101.07202*, 2021.

# Chapter 13

# Paper F: Model Checking Collision Avoidance of Nonlinear Autonomous Vehicle Models

Rong Gu, Cristina Seceleanu, Eduard Enoiu, and Kristina Lundqvist. Published in Proceedings of the 24th International Symposium on Formal Methods, Springer, 2021.

## Abstract

Autonomous vehicles are expected to be able to avoid static and dynamic obstacles automatically, along their way. However, most of the collision-avoidance functionality is not formally verified, which hinders ensuring such systems' safety. In this paper, we introduce formal definitions of the vehicle's movement and trajectory, based on hybrid transition systems. Since formally verifying hybrid systems algorithmically is undecidable, we reduce the verification of nonlinear vehicle behavior to verifying discrete-time vehicle behavior overapproximations. Using this result, we propose a generic approach to formally verify autonomous vehicles with nonlinear behavior against reach-avoid requirements. The approach provides a UPPAAL timed-automata model of vehicle behavior, and uses UPPAAL STRATEGO for verifying the model with user-programmed libraries of collision-avoidance algorithms. Our experiments show the approach's effectiveness in discovering bugs in a state-of-the-art version of a selected collision-avoidance algorithm, as well as in proving the absence of bugs in the algorithm's improved version.

## 13.1 Introduction

Autonomous vehicles (AV), such as driverless cars and robots, are becoming increasingly promising, hence prompting a wide interest in industry and academia. Safety of vehicle operations is the most important concern, requiring these systems to move and act without colliding with static or dynamic objects (obstacles) in the environment, such as big rocks, humans, and other mobile machines. Algorithms like A* [1], Rapidly-exploring Random Tree (RRT) [2], and Theta* [3] are able to navigate the AV towards reaching their destinations, while avoiding static obstacles along the way. However, when encountering dynamic obstacles that could appear and move arbitrarily in the environment, these algorithms are not enough for collision avoidance, and have to be complemented by algorithms such as those based on dipole flow fields [4] or dynamic window approach [5], which are capable of circumventing dynamic obstacles.

Although many collision-avoidance algorithms are being proposed in recent years, few of them have been formally verified, despite the fact that formal verification is a very important tool for discovering problems in the early stage of algorithm design. In this paper, we consider two main challenges that can turn formal verification of AV models and their algorithms into a daunting task: (i) nonlinearity of the vehicle kinematics, and (ii) complexity and uncertainty of the environment where AV move. On the one hand, ordinary differential equations are used to describe the continuous dynamics and kinematics of the often nonlinear vehicles. The trajectories formed by these vehicle models are consequently nonlinear, which is the nonlinearity that we consider throughout the paper. On the other hand, discrete decisions made by the vehicles' control systems influence the movement of vehicles. In the model-checking world, verification of these so-called *nonlinear hybrid systems* that combine nonlinear continuous kinematics and discrete control is undecidable [6, 7]. In addition, AV that aim at tracking initially planned paths are inevitably diverted by their tracking errors caused by the inaccuracy of their sensors and actuators, and the disturbance from the complex environment. Dynamic obstacles are unpredictable before AV sense them. All these reasons render exhaustive model checking of models of nonlinear vehicles that move in an environment containing static obstacles and uncertain dynamic obstacles an unsolved problem.

In this paper, we solve this problem by addressing challenges (i) and (ii). First, we introduce *safe zones* of the trajectories formed by *controllable* nonlinear AV models, which overcomes challenge (i), as follows. If an AV's tracking error has a Lyapunov function, it is called *controllable* in this paper, and its deviation from the reference path is bounded [8]. The boundaries of tracking errors form the safe zone of the AV, assuming the reference path as the axis. As long as the dynamic obstacles do not intrude into these zones, the vehicles are guaranteed to be safe. Based on this observation, we reduce the verification of *controllable* AV's nonlinear trajectories to the verification of its piece-wise-continuous (PWC) reference trajectories, and further to the verification of discrete-time models of trajectories. The various vehicle dynamics and kinematics, together with the uncertain tracking errors are all subsumed by the safe zones, so the undecidable verification problem is simplified to a decidable one, without losing completeness.

Next, we solve challenge (ii) by leveraging the nondeterminism of timed automata in UPPAAL STRATEGO [9]. The initialization and movement of dynamic obstacles are modeled as timed automata, in which their positions etc. are nondeterministically initialized and updated. In this way, the vehicle model satisfies the *liveness* property only when it is able to reach the destination, and the *invariance* property if there is no collision happening under any circumstance. When multiple dynamic obstacles are involved, the state space of the model becomes large and the verification becomes computationally expensive or even unsolvable. Consequently, we also propose a way of reducing the state space by splitting the verification into multiple tractable phases.

Note that, our approach is orthogonal to the methods of controller synthesis (e.g., [10, 11]). The latter targets the construction of motion plans that avoid static and dynamic obstacles, whereas our method can be used to verify the correctness of these methods, regardless of the path-planning and collision-avoidance algorithms considered. To summarize, our main contributions are:

1. A proven transformation of the verification of nonlinear vehicle trajectories to the verification of PWC trajectories and discrete-time trajectories.

2. A generic verification approach for model checking reach-avoid requirements of AV equipped with different collision-avoidance algorithms (Section 13.4).

3. An implementation of the approach in UPPAAL STRATEGO, and a demonstration showing the ability of the approach to discover bugs in a state-of-the-art collision-avoidance algorithm, and to prove the absence of bugs in an improved version of the same algorithm (Section 13.5).

**Preliminaries**. In this paper, we denote a vector $x$ by $\vec{x}$, the module of $\vec{x}$ by $||\vec{x}||$, and multiplications between two scalars, and between a vector and a scalar by "$\times$". *Timed Automata* is a widely-used formalism for modeling real-time systems [12]. The UPPAAL model checker [13] uses an extension of the timed-automata language with a number of features such as constants, data variables, arithmetic operations, arrays, broadcast channels, urgent and committed locations. Properties that can be checked by UPPAAL are formalized in a simplified *timed computation tree logic* (TCTL) [14], which basically contains a decidable subset of *computation tree logic* (CTL) plus clock constraints. A branch of UPPAAL, named UPPAAL STRATEGO [9], supports calling external C-code functions written in libraries. This new feature enables us to treat the user-designed collision-avoidance algorithm as a black box in our model.

The remainder of the paper is organized as follows. In Section 13.2, we introduce the systems to be verified. In Section 13.3, we concretely define the movement and trajectories of AV and prove two theorems of transforming the verification of nonlinear vehicle trajectories to the verification of PWC trajectories and discrete-time trajectories. A detailed description of the verification approach and tool support is presented in Section 13.4, followed by experiments in Section 13.5. We compare our study to related work in Section 13.6, and conclude the paper in Section 13.7.

## 13.2 Problem Description

Vehicles that are capable to calculate paths to their destinations, which avoid collision with any obstacles in the environment, and follow them without human intervention, are called *autonomous vehicles* (AV). As depicted in Fig. 13.1, when the environment contains only static obstacles whose positions are already known by the AV, paths are calculated by the path planner inside the controller of the AV. Path planners are usually equipped with path-planning algorithms, e.g., Theta* [3] or

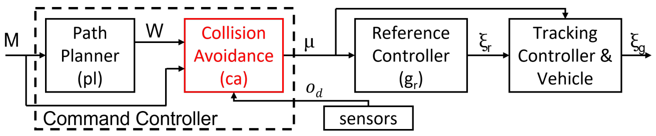


Figure 13.1: The architecture of the controller of autonomous vehicles. The collision-avoidance module does not exist if the environment only contains static obstacles.

RRT [2], which explore the map ($M$) to find a path that avoids the static obstacles and reaches the destination. The reference controller ($g_r$) uses the output of the path planner and generates a trajectory of the state variables of the system, e.g., position and linear velocity of the vehicle, as a reference ($\xi_r$) for the tracking controller to follow. The tracking controller aims to produce an input to the vehicle to drive it to track the reference trajectory. The real trajectory ($\xi_g$) follows the reference path ($\xi_r$) with some tracking errors.

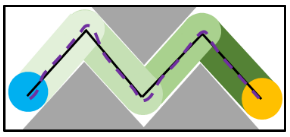


Figure 13.2: The reference trajectory is solid black lines, and the actual trajectory is violet dotted lines. The initial area is blue and the goal area is yellow. The boundaries of tracking errors are green. Static obstacles are grey [8].

Since the dynamics and kinematics of a real AV are nonlinear, and tracking errors between the actual trajectory and reference trajectory exist inevitably, path planners do not guarantee the safety of AV driving. Moreover, formally verifying if the actual trajectories ever hit the static obstacles is an undecidable problem, due to the model-checking of nonlinear hybrid systems being undecidable [7]. Overapproximation is a method of linearizing the vehicle model, to facilitate verification. Fan et al. [8] propose a method that proves that, as long as the dynamics of tracking errors has a Lyapunov function, the tracking errors are bounded by a piece-wise constant value, which depends on the initial tracking error and the number of segments of the reference trajectory. Fig. 13.2 shows an example of a reference trajectory and the boundary of tracking errors. Consequently, as long as the safe regions of AV (green color) do not overlap with the grey areas, the actual trajectory is guaranteed to be safe.

Due to this result, one can reduce the problem of verifying whether the actual trajectory ($\xi_g$) ever overlaps with obstacles, to a simplified problem of verifying whether the distance between the reference trajectory ($\xi_r$) and the obstacles is larger than the respective boundary of tracking error on each segment of $\xi_r$. In other words, the verification of nonlinear vehicle trajectories is reduced to the verification of their piece-wise-continuous reference trajectories. Although much simplified, the problem is still undecidable as long as the piece-wise-continuous trajectories are non-linear [8]. Moreover, when dynamic obstacles appear, the verification becomes intractable, because dynamic obstacles cannot be known completely before the AV encounters them. The controller must be additionally equipped with a collision-avoidance module that perceives the environment periodically, via sensors. Fig. 13.1 shows such a controller. The path planner still calculates a path that avoids known static obstacles and goes to the destination. The path serves as input to the collision-avoidance module as a sequence of waypoints (positions of turning directions, denoted as $W$), as well as the information of the map ($M$) and dynamic obstacles ($o_d$). The command controller should meet the following two requirements, which are the focus of verification in this paper:

- Collision avoidance (invariance property): always circumventing the static and dynamic obstacles;
- Destination reaching (liveness property): always eventually reaching the goal area.

## 13.3 Definitions and Verification Reduction Theorems

In this section, we introduce the definitions of the important concepts used in this paper and the collision-avoidance verification theorems that eventually reduce the nonlinear trajectory verification to discrete-time trajectory verification. We denote AV and dynamic obstacles collectively by the term *agents*.

First, let us establish an overall view of the different types of models that are used in this section. So far, we have stated that model-checking *liveness* properties (e.g., destination reaching) and *invariance* properties (e.g., collision avoidance) of nonlinear hybrid systems is undecidable. Note that hybrid systems are described by syntactic models with an underlying semantics defined as hybrid transition systems (HTS), used in the following definitions. As depicted in Fig. 13.3, the contin-

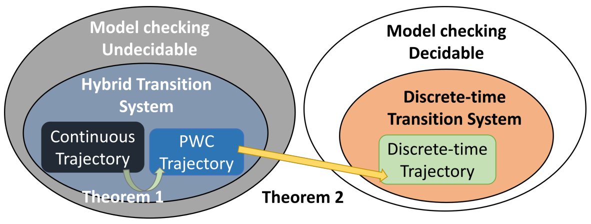


Figure 13.3: Overall description of models and their decidability

uous trajectories of agents are modeled as HTS. By incorporating the tracking errors of agents, the continuous trajectories are simplified into piece-wise-continuous (PWC) trajectories. However, the verification of PWC trajectories is still undecidable, so we transform the PWC trajectories into discrete-time trajectories, whose verification is decidable. Furthermore, the two-step transformation from continuous trajectories to discrete-time trajectories is proved to preserve the *liveness* and *invariance* properties that we want to verify (Theorem 3 and Theorem 4).

## 13.3.1 Definitions of Maps, Agent States, and Trajectories

In this section, we first define the agent states and the map where agents move. Next, we define the command controllers and agent-state trajectories.

**Definition 17** (Map)**.** *A map is a 4-tuple $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$, where (i) $\mathcal{X} \in \mathbb{R}^d$ is the moving space, with $d \in \{2, 3\}$ being the dimension of the map, (ii) $\boldsymbol{O}_u \subseteq \mathcal{X}$ is the unsafe area, (iii) $\mathcal{I} \subseteq \mathcal{X}$ is the initial area of AV, and (iv) $\mathcal{G} \subseteq \mathcal{X}$ is the goal area where the AV aims to go.*

An example of a map is illustrated in Fig. 13.2.

**Definition 18** (Agent State)**.** *Given a map $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$, an agent state is a 5-tuple $\mathcal{S} =< \vec{p}, \vec{v}, \vec{a}, \theta, \omega >$, where (i) $\vec{p} \in \mathcal{X}$ is the position vector, (ii) $\vec{v}$ is the linear velocity vector, $||\vec{v}|| \in [0, V_{max}] \subset$*

*$\mathbb{R}_{\geq 0}$, (iii) $\vec{a}$ is the acceleration vector, $||\vec{a}|| \in [A_{min}, A_{max}] \subset \mathbb{R}$, (iv) $\theta \in [-\pi, \pi] \subset \mathbb{R}$ is the heading, and (v) $\omega \in [\Omega_{min}, \Omega_{max}] \subset \mathbb{R}$ is the rotational velocity.*

The agent states are states of AV and dynamic obstacles. Some elements in the tuple of agent states $\mathcal{S}$ evolve continuously and some are assumed to change instantaneously. We define the trajectories of the evolution of the agent states in Definition 20. Before that, we first define the controller of AV, where dynamic obstacles ($\mathbf{O}_d$) are instances of agent states $\mathcal{S}$, as follows:

**Definition 19** (Controller)**.** *Given a map $\mathcal{M}$, and a set of dynamic obstacles $\boldsymbol{O}_d$, we define a command controller of AV as a 3-tuple $\mathcal{C} =< pl, ca, \Lambda >$, where (i) $pl : \mathcal{M} \longrightarrow \mathcal{W}$ is a path-planning function, $\mathcal{W} \subseteq \mathcal{X}$ is a set of waypoints, (ii) $ca : \mathcal{M} \times \mathcal{W} \times \boldsymbol{O}_d \longrightarrow \Lambda$ is a collision-avoidance function, and (iii) $\Lambda = \{ACC, BRK, TR^+, TR^-, STR\}$ is a set of commands.*

The commands are signals sent from the controllers to the actuators of the AV: $ACC$ means acceleration, $BRK$ means brake, $TR^+$ and $TR^-$ mean turning counter-clockwise and clockwise, respectively, and $STR$ means moving straightly at a constant speed. An example of the AV's controller architecture is shown in Fig. 13.1. When an AV starts to move, the transitions of its agent states form a trajectory, in which its position, linear velocity, and heading evolve continuously according to corresponding dynamic functions, whereas its acceleration and rotational velocity change discretely based on the commands.

**Definition 20** (Continuous Trajectory)**.** *Given an AV, whose command controller is $\mathcal{C} =< pl, ca, \Lambda >$, we define its movement by a hybrid transition system $< S, s_0, \Sigma, X, \rightarrow>$, where $S$ is a set of states, $s_0$ is the initial state, $\Sigma \subseteq \Lambda$ is the alphabet, $X = X_d \cup X_c$ is a set of variables combining discrete variables in $X_d$ and continuous variables in $X_c$, and $\rightarrow$ is a set of transitions defined by the following rules, with kinematic functions of the AV denoted by $f$:*

- *Delayed transitions: $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{\Delta t} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $t \in X_c$, $\vec{p}' = \vec{p} + \int_l^u \vec{v} dt$, $\vec{v}' = \vec{v} + \int_l^u \vec{a} dt$, $\vec{a}' = \vec{a}$, $\theta' = \theta + \int_l^u \omega dt$, $\omega' = \omega$, $l \in \mathbb{R}_{\geq 0}$ and $u \in \mathbb{R}_{>0}$ are the upper and lower time bounds, respectively, and $\Delta t = u - l$;*

- *Instantaneous transitions:* $<\vec{p},\vec{v},\vec{a},\theta,\omega>\xrightarrow{cmd}<\vec{p}',\vec{v}',\vec{a}',\theta',\omega'>$, *where* $\vec{p}'=\vec{p}$, $\vec{v}'=\vec{v}$, $\vec{a}'=ca(\vec{a},cmd)$, $\theta'=\theta$, $\omega'=ca(\omega,cmd)$, $cmd\in\Sigma$.

A run of the transition system defined above over a duration $U$ is a *trajectory* of agent states, also described by the function $\xi:[0,U]\rightarrow\mathcal{S}$. Henceforth, we name the agent-state trajectory as *trajectory* for brevity, and denote $\xi(t)$ as a point of $\xi$ at time $t$, the projection of $\xi$ on a dimension of an agent-state as $\xi\downarrow dimension$, e.g., positions on a trajectory are $\xi\downarrow\vec{p}$. The continuous variables of actual trajectories of agents are generated by their nonlinear kinematic functions, yet these variables are piece-wise-continuous (PWC) in reference trajectories (see Fig. 13.2). More specific, a reference trajectory $\xi_r$ is a sequence of concatenated trajectory segments $\xi_{r,1}\frown...\frown\xi_{r,k}$. The concatenating points $\{\vec{p}_i\}_{i=0}^{k}$ are the waypoints calculated by path-planners, where the discontinuity of the vehicle's heading $\theta$ happens. Therefore, the definition of agent movement on a reference trajectory changes as follows:

**Definition 21** (Reference Trajectory)**.** *Let us assume an AV, whose command controller is* $\mathcal{C}=<pl,ca,\Lambda>$*, and a PWC trajectory* $\xi_r$ *of the AV, which is a sequence of trajectories* $\xi_{r,1}\frown...\frown\xi_{r,k}$ *concatenated by a set of waypoints* $\{\vec{P}_i\}_{i=0}^{k}$*. Then, the AV's movement along the reference trajectory is a hybrid transition system similar to that of Definition 20, and its transitions are defined by the following rules:*

- *Delayed transitions on* $\xi_r\downarrow\vec{p}\not\subseteq\{\vec{P}_i\}_{i=0}^{k}$: $<\vec{p},\vec{v},\vec{a},\theta,\omega>\xrightarrow{\Delta t}$ $<\vec{p}',\vec{v}',\vec{a}',\theta',\omega'>$, *where* $\vec{p}'=\vec{p}+(\vec{v}+\frac{\vec{a}\times\Delta t}{2})\times\Delta t$, $\vec{v}'=\vec{v}+\vec{a}\times\Delta t$, $\vec{a}'=\vec{a}$, $\theta'=\theta$, $\omega'=0$*;*

- *Instantaneous transitions:* $<\vec{p},\vec{v},\vec{a},\theta,\omega>\xrightarrow{cmd}<\vec{p}',\vec{v}',\vec{a}',\theta',\omega'>$, *where* $\vec{p}'=\vec{p}$, $\vec{v}'=\vec{v}$, $\vec{a}'=ca(\vec{a},cmd)$,
$$\theta'=\begin{cases} arctangent(\vec{P}_i,\vec{P}_{i+1}),\ \ if\ \vec{p}\in\{\vec{P}_i\}_{i=0}^{k-1}, \\ \theta,\ \ if\ \vec{p}\notin\{\vec{P}_i\}_{i=0}^{k-1}\end{cases},\ \omega'=0$$

Intuitively, when an agent is moving along its reference trajectory ($\xi_r$), its heading ($\xi_r\downarrow\theta$) remains unchanged before it arrives at a waypoint, which means the rotational velocity ($\xi_r\downarrow\omega$) is irrelevant and remains 0. Therefore, the reference trajectory is infeasible to be tracked exactly by the agents. Although the integration of $\xi_r\downarrow\vec{p}$ and $\xi_r\downarrow\vec{v}$ on delayed transitions is simplified to polynomial functions, the nonlinearity of

$\xi_r{\downarrow}\vec{p}$ still renders undecidability. The trigonometric function in the definition also causes a computational difficulty when running verification. In practice, we use linear speed vector ($\vec{v}$) to describe both the linear speed and the orientation of the agent. The acceleration ($\xi_r{\downarrow}\vec{a}$) changes instantaneously based on the commands from the command controller. Last but not least, the trajectories of dynamic obstacles are similar to Definition 20, but without a well-defined controller. On their instantaneous transitions, accelerations and rotational velocities are changed arbitrarily within the valid ranges.

## 13.3.2 Collision-Avoidance Verification Reduction

We use $\xi_r$ and $\xi_g$ to denote the reference and actual trajectory of AV, respectively, and $\xi_o$ for the actual trajectories of dynamic obstacles.

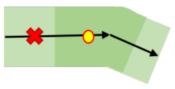


Figure 13.4: A dynamic obstacle is at the red cross, while the current position of AV on the reference path is the yellow dot. The safety-critical area is dark green.

Let $d(var_1, var_2)$ denote the distance between $var_1$ and $var_2$, e.g., $d(\vec{p}_i, \xi_j{\downarrow}\vec{p})$ is the distance from position $\vec{p}_i$ to trajectory $\xi_j{\downarrow}\vec{p}$, and $d(\xi_i{\downarrow}\vec{p}, \mathbf{O}_u)$ is the distance from trajectory $\xi_i{\downarrow}\vec{p}$ to static obstacles. For brevity, we omit the projection when using this notation, i.e., $d(\vec{p}_i, \xi_j{\downarrow}\vec{p}) = d(\vec{p}_i, \xi_j)$. Let $\xi(t_1, t_2)$ denote a segment of trajectory $\xi$ between time points $t_1$ and $t_2$. The problem of verifying if AV hit static obstacles $\mathbf{O}_u$ is relatively simple, as $\mathbf{O}_u$ does not change. However, checking if AV hit moving obstacles is different and much harder, because both trajectories are formed dynamically while the agents are moving. Dynamic obstacles might meet an AV's reference trajectory, yet far enough from its current position(see Fig. 13.4). Therefore, we introduce the concept of *safety-critical segments*:

**Definition 22** (Safety-Critical Segment)**.** *Let $C$ be the current time. Given a trajectory $\xi$, a time span of length $T \in \mathbb{R}_{>0}$, we define a safety-critical segment $sc(\xi)$ of $\xi$, as $\xi(C-T, C+T)$*[1]*.*

The length of time-span $T$, so that the safety-critical area covers the actual current position of AV, can be delivered by design engineers

[1]When $C < T$, $sc(\xi) = \xi(0, C+T)$.

with knowledge of vehicle dynamics, so this is not within the scope of this paper. Now, instead of checking if any part of the AV's entire trajectory ($\xi_g$) overlaps with a moving obstacle's trajectory ($\xi_o$), we check if the safety-critical segments of these two trajectories ($sc(\xi_g)$ and $sc(\xi_o)$) overlap.

**Definition 23** (Collision-Avoidance Verification). *Given a map $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$, a nonlinear AV, whose actual continuous trajectory is $\xi_g$, and a set of dynamic obstacles whose trajectories are in set $\Xi_o$, we say that the collision-avoidance verification of the AV's actual trajectory equates with verifying that condition $\xi_g{\downarrow}\vec{p} \cap \mathcal{G} \neq \emptyset \wedge \xi_g{\downarrow}\vec{p} \cap \boldsymbol{O}_u = \emptyset \wedge sc(\xi_g{\downarrow}\vec{p}) \cap sc(\xi_o{\downarrow}\vec{p}) = \emptyset$ holds, where $\xi_o \in \Xi_o$.*

Since model-checking $\xi_g$ is undecidable, we prove next that its verification can be reduced to one over the PWC trajectory $\xi_r$ that $\xi_g$ tracks.

**Theorem 3** (Non-linearity to PWC). *Assume the collision-avoidance verification condition of Definition 23, a position $\vec{p}_g \in \mathcal{G}$ whose distance to the closest boundary of $\mathcal{G}$ is $B$, and that the tracking errors of the AV have a Lyapunov function. Then, it follows that if the condition $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \emptyset \wedge d(\xi_r, \boldsymbol{O}_u) > L \wedge d(sc(\xi_r), sc(\xi_o)) > L$, with $L \in \mathbb{R}_{>0}$ and $L \leq B$ holds, then the collision-avoidance condition of Definition 23 holds too.*

*Proof.* Based on Lemmas 2 and 3 proven by Fan et al. [8], if the tracking errors of the AV have a Lyapunov function, its $\xi_g$ is bounded within a certain distance to its $\xi_r$. Let the distance be $L$, then $d(\xi_g, \xi_r) < L \leq B$. Hence, if $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \emptyset$, then $\xi_g{\downarrow}\vec{p} \cap \mathcal{G} \neq \emptyset$. Since $d(\xi_r, \mathbf{O}_u) > L > d(\xi_g, \xi_r)$ and $d(sc(\xi_r), sc(\xi_o)) > L > d(\xi_g, \xi_r)$, then $\xi_g{\downarrow}\vec{p} \cap \mathbf{O}_u = \emptyset \wedge sc(\xi_g{\downarrow}\vec{p}) \cap sc(\xi_o{\downarrow}\vec{p}) = \emptyset$. □ □

Note that these two problems are not equivalent. When the actual trajectory is not colliding with any obstacles, the distance from the reference trajectory to the obstacles could be less than $L$. The method of calculating $L$ is not the concern of this paper. We refer the reader to literature [8] for details.

### 13.3.3 Discretization of Trajectories

Although the verification of nonlinear trajectories is simplified by Theorem 3, model-checking PWC trajectories is still difficult. PWC trajectories are described by hybrid systems, in which variables, e.g., $\vec{p}$ and

$\vec{v}$, change continuously (specifically, $\vec{p}$ is nonlinear), whereas variables, e.g., $\theta$, $\vec{a}$ and $\omega$, change instantaneously (Definition 21). Unfortunately, the algorithmic verification of such model is undecidable [15]. To make the problem tractable, we discretize PWC trajectories into a discrete-time model, where the movement of agents (including AV and dynamic obstacles) is sampled synchronously:

**Definition 24** (Discrete-Time Trajectory). *Given a PWC trajectory named $\xi_r$, whose concatenating points (waypoints) are $\{\vec{P}_i\}_{i=0}^{k}$, a discretized trajectory $\xi_{rd}$ of $\xi_r$ is a run of a corresponding discrete-time transition system $< D, d_0, \Pi, \rightarrow >$, where $D$ is the set of states, $d_0$ is the initial state, $\Pi \subseteq \Lambda \cup \{sync\}$ is the set of labels consisting of controller commands and a label for synchronization with other discretized trajectories, and $\rightarrow$ is a transition relation, in which the instantaneous transitions of $\theta, \vec{a}$ and $\omega$ remain the same as defined in Definition 21, and the delayed transitions are sampled at the time points when $\Delta t = \varepsilon$, where $\varepsilon \in \mathbb{R}_{>0}$ is the granularity of sampling:*

- *if $\Delta t < \varepsilon$, $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega >$ does not change,*
- *if $\Delta t = \varepsilon$, $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{\Delta t, sync} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $\theta' = \theta, \omega' = \omega, \vec{a}' = \vec{a}, \vec{v}' = \begin{cases} \vec{v} + \vec{a} \times \varepsilon, \ \textit{if}\ ||\vec{v} + \vec{a} \times \varepsilon|| < V_{max}, \\ \frac{\vec{v}}{||\vec{v}||} \times V_{max}, \ \textit{if}\ ||\vec{v} + \vec{a} \times \varepsilon|| \geq V_{max} \end{cases}$,*
  $\vec{p}' = \begin{cases} \vec{P}_i, \ \textit{if}\ \vec{p} + (\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon \succ \vec{P}_i, \\ \vec{p} + (\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon, \\ \quad \textit{if}\ \vec{p} + (\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon \preccurlyeq \vec{P}_i \end{cases}$

To denote if the position passes (resp., does not pass) the next waypoint, we use the syntactic sugar $\succ$ (resp., $\preccurlyeq$). The algorithm of judging this is given in literature [16]. Intuitively, when the time interval $\Delta t$ is less than a small period $\varepsilon$, the environment is not observed, so the trajectories of the agents are not sampled; when $\Delta t$ reaches $\varepsilon$, the agent states are observed and sampled. When an agent reaches or passes its target waypoint in the current period $\varepsilon$, it stops at the waypoint until the next period comes when the new waypoint and heading are updated by the instantaneous transitions.

Dynamic obstacles do not have pre-computed waypoints but appear and move arbitrarily in the map. However, a reasonable obstacle would not change its direction too frequently, e.g., every sampling period. We design dynamic obstacles such that, initially, they choose their starting agent-states arbitrarily. Then, they keep moving for $N$ sampling periods before choosing a new agent-state as a target. The straight path between the current and target positions is a reference trajectory that the dynamic obstacle tracks in the next $N$ periods, and the tracking errors are also bounded.

The agents' accelerations and rotational velocities are assumed to be changing discretely in these definitions. If the assumption is violated in some applications, one can discretize these two variables in the same way as in the discretization of position and linear velocity. Next, we prove a theorem that reduces the verification of PWC reference trajectories to the one of discrete-time trajectories.

**Theorem 4.** *(PWC to discrete-time trajectories). Assume a map* $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$*, a set of trajectories* $\Xi_o$ *formed by dynamic obstacles, with the maximum linear velocity* $V$*, a reference trajectory* $\xi_r$ *of an AV with concatenating points* $\{\vec{P}_i\}_{i=0}^{k}$*, whose safety-critical segment is* $sc(\xi_r)$*, and synchronized and discretized trajectories* $\xi_{rd}$ *of* $\xi_r$*, and* $\xi_{od}$ *of* $\xi_o \in \Xi_o$ *with a granularity of sampling* $\varepsilon \leq \frac{L}{||V||}$*; here,* $L = L_a + L_o$*, where* $L_a$ *is the tracking-error boundary of the AV, and* $L_o$ *is the smallest tracking-error boundary among dynamic obstacles*[2]*. Then, if* $\vec{p}_g \in \mathcal{G}$*, and* $\xi_{rd}{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \emptyset \wedge d(\xi_{rd}, \boldsymbol{O}_u) > L \wedge d(sc(\xi_{od}), sc(\xi_r)) > L$*, it follows that* $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \emptyset \wedge d(\xi_r, \boldsymbol{O}_u) > L \wedge d(sc(\xi_o), sc(\xi_r)) > L$*.*

*Proof.* By substituting $\Delta t$ in the delay transitions of Definition 21 with $\varepsilon$, we can see that $\xi_{rd}(\varepsilon)$ is a sampling of the reference trajectory $\xi_r(t)$ at the time points when $\Delta t = \varepsilon$. Hence, $\xi_{rd}{\downarrow}\vec{p} \subseteq \xi_r{\downarrow}\vec{p}$. Therefore, if $\xi_{rd}{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \emptyset$, which means $\xi_{rd}$ can reach $\vec{p}_g$, then $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \emptyset$ as well.

Based on Definition 24, waypoints $\{\vec{P}_i\}_{i=0}^{k} \subseteq \xi_{rd}{\downarrow}\vec{p}$, where turning occurs. Therefore, if $t_i$ and $t_{i+1}$ are two consecutive sampling points of $\xi_{rd}$, the line segment connecting $t_i$ and $t_{i+1}$ must be on $\xi_r$, denoted by $\xi_{rd}(t_i, t_{i+1})$. Therefore, if $d(\mathbf{O}_u, \xi_{rd}(t_i, t_{i+1})) > L$[3], then the concatenation of $\{\xi_{rd}(t_i, t_{i+1})\}_{i=0}^{n-1}$, which is $\xi_r$, satisfies $d(\mathbf{O}_u, \xi_r) > L$.

For $\xi_o \in \Xi_o$, similarly, $t_i$ and $t_{i+1}$ are two consecutive sampling points. As depicted in Fig. 13.5, $\xi_o(t_i,\ t_{i+1})$ and $\xi_r(t_i,\ t_{i+1})$ are the segments of $sc(\xi_o)$ and $sc(\xi_r)$, respectively. Assume $d(sc(\xi_{od}),\ sc(\xi_r)) > L$, but $d(sc(\xi_o),\ sc(\xi_r)) \leq L$, which means $d(\xi_{od}(t_i),\ \xi_r(t_i, t_{i+1})) > L$

[2]When no dynamic obstacle is detected, $L_o$ is zero.

[3]Computation of $d(\mathbf{O}_u, \xi_{rd}(t_i, t_{i+1}))$ is in a more detailed version of this paper [16].

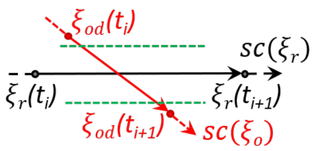


Figure 13.5: The trajectory of a dynamic obstacle is red. The reference trajectory of AV is black. Dotted greens lines are the boundaries of tracking errors.

and $d(\xi_{od}(t_{i+1}), \xi_r(t_i,\ t_{i+1})) > L$, but $d(\xi_o(t_i,\ t_{i+1}), \xi_r(t_i,\ t_{i+1})) \leq L$, then $\xi_o(t_i,\ t_{i+1})$ and $\xi_r(t_i,\ t_{i+1})$ must be intersecting, and thus $d(\xi_o(t_i),\ \xi_o(t_{i+1})) > L$ (see Fig. 13.5). Based on Definition 24, $d(\xi_o(t_i),$ $\xi_o(t_{i+1})) = ||(\vec{v} + \frac{\vec{a}\times\varepsilon}{2}) \times \varepsilon|| \leq ||V|| \times \varepsilon$. Therefore, $||V|| \times \varepsilon > L$, which contradicts the assumption $\varepsilon \leq \frac{L}{||V||}$. Hence, if $d(sc(\xi_{od}),\ sc(\xi_r)) > L$, then $d(sc(\xi_o),\ sc(\xi_r)) > L$. □

Based on Theorems 1 and 2, the reach-avoid verification of discretized trajectories is sufficient to entail that of nonlinear trajectories. The reach-avoid verification of discrete-time transition systems is decidable [6]. Therefore, the undecidable problem of model-checking nonlinear trajectories of agents is successfully simplified to a decidable one over discrete-time trajectories. In the next section, we introduce our approach of verifying the discrete-time models.

## 13.4 Verification Approach and Tool Support

In our verification approach, we employ UPPAAL Timed Automata (UTA) [13] to build the discrete-time model of the agents, and UPPAAL STRATEGO as the model checker to execute the verification. The latest version of UPPAAL STRATEGO provides a function of calling external libraries. This function enables us to design a model for verification without knowing the implementation details of algorithms, hence modeling them as black boxes. Although UPPAAL STRATEGO is mainly designed for strategy synthesis of stochastic timed games, our approach only leverages its function of exhaustive model checking. The semantics of UTA is timed transition systems. When discretizing time in timed transition systems, one gets discrete-time transition systems, which can be used to model the discrete-time trajectory of agents (Definition 24). Our UTA templates are designed to act only at the end of each sampling period simultaneously, so within the sampling periods, nothing happens but only time elapses. Therefore, the semantics of our UTA templates is shown to be conservatively abstracted by the discrete-time transition semantics, with the discretizing step being equal to the sampling period of the discrete-time trajectories.

### 13.4.1 General Description of the Approach

Fig. 13.6 shows the workflow of the verification approach. The input of the approach is the parameters of the agents (i.e., AV and dynamic obstacles) and their boundary of the tracking errors, as well as the environment (e.g., static obstacles). In Step 1, users provide their nonlinear

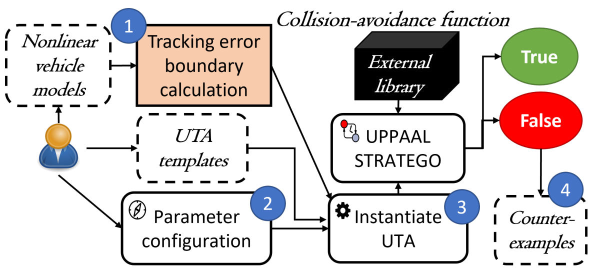


Figure 13.6: The workflow of the verification approach

vehicle models, which are for calculating the boundary of tracking errors. This module is the approach provided by Fan et al. [8], which is not the focus of this paper. We simply use the output of this approach in our models for verification. In Step 2, users configure the parameters of the approach, which are used for instantiating the UTA models. Parameters regulate the minimum and maximum values of the elements of agent states, e.g., linear velocity. The detailed specification of the parameters is in literature [16]. In Step 3, UTA templates of the discrete-time models are instantiated into UTA models based on the configured parameters. Note that the user-programmed collision-avoidance algorithm is embedded in the models as executable libraries, e.g., Dynamic-Link Libraries (DLL) in Windows, or Shared Object (SO) in Linux. After the instantiation of UTA, the model checker verifies the model by traversing its state space, calling the external libraries when necessary, and checking if the vehicle model avoids all obstacles and reaches the destination under all circumstances. If the verification result is "true", the algorithm is guaranteed to be correct under the current parameter configuration; otherwise, counter-examples are returned by the model checker for the users to debug their algorithm or change the configuration of the parameters (Step 4).

## 13.4.2 Design of the UTA Templates and CTL Properties

There are four UTA templates that are well designed to be reusable. The figures and the detailed description of the templates are in our technical report [16]. First, we overview the UTA templates:

- **AV Parameter Template**. Based on Definition 24, after being initialized, the AV parameters (e.g., position, speed) either stay unchanged or update their values at the end of the sampling periods, simultaneously. Therefore, we define this template for updating the AV parameters periodically. Instances of this template are parameters of AV, hence, users can add their parameters of AV simply by instantiating this template. The update of AV parameters are synchronized by the controller template.

- **AV Controller Template**. The AV controller template mainly accomplishes three jobs: initializing the AV parameters; invoking the UTA of AV parameters periodically; making decisions, such as turning at waypoints, or calling the external function of collision avoidance when seeing an obstacle.

- **Obstacle Initialization Template**. As depicted by its name, this template is responsible for initializing moving obstacles. For each parameter of the obstacle (e.g., position, speed), the template traverses the range of its value and nondeterministically chooses one to be the initial value of the parameter. Therefore, when running the exhaustive model checking in UPPAAL STRATEGO, all the values are enumerated and verified.

- **Obstacle Movement Template**. This template is for updating the obstacle's parameters periodically. At every end of the sampling period, the AV controller UTA invokes the AV parameter UTA as well as the obstacle movement UTA. In this way, sampling the AV and dynamic obstacles is synchronized at the same moments. Note that this template updates the acceleration and heading of the obstacle every $N$ periods, $N > 1$. As aforementioned, reasonable obstacles do not change their direction and acceleration too frequently.

The CTL properties that formalize the reach-avoid requirement are as following:

- **Obstacle avoiding**: `A[]!collision`, where `collision` is a Boolean variable that is updated every sampling period. When the distances from the safety-critical segment of AV to any of the obstacles in the map is less than the boundaries of tracking errors, `collision` is turned to true, and remains *false* elsewhere. Therefore, this query asks: for *all* execution paths, is collision *always* avoided?

- **Destination reaching**: `A<>controller.STOP`, where `STOP` is a location in UTA of AV's controller. When `controller` goes to location *STOP*, it means that the AV has reached the destination. Therefore, this query asks: for *all* the execution paths of the model, does AV *eventually* reach the destination?

### 13.4.3 Reduction of the State Space of the UTA Model

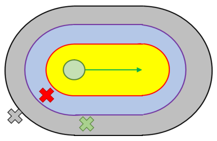


Figure 13.7: The green arrow is the reference path. The green circle is the AV. The crosses are the dynamic obstacles, where red and grey ones are invalid positions, and the green cross is valid.

To explore all the possible behaviors of dynamic obstacles, in the worst-case scenario, we would have to explore the entire map, and enumerate all possible values of linear speeds, rotational speeds, and headings of dynamic obstacles. This generates a huge state space of the model that can be infeasible to check. In this section, we introduce how to reduce the state space of the UTA model without damaging the completeness of the verification.

**Reduction of Initial Values of Parameters.** Even though the dynamic obstacles can appear at any positions in the map, some positions are too far away from the AV to be relevant at the current period, and some are too close to the AV to be possible to be avoided. Hence, we categorize positions into three classes, namely *safety-critical* area, *closest* area, and *valid* area. Fig. 13.7 depicts these three kinds of areas. The safety-critical area is defined in Definition 22.

Positions from which the distance to the safety-critical segment of the reference path is shorter than or equal to $V \times n \times \varepsilon$ is called *closest* area, where $V$ is the velocity of the dynamic obstacle, $\varepsilon$ is the sampling period, and $n \in \mathbb{N}$ is a coefficient whose value depends on the physical limitations of the AV. Obstacles appearing within the closest area are impossible to be avoided, so they should be excluded from the valid initial positions. Similarly, positions from which the distance to the safety-critical segment is greater than $V \times n \times \varepsilon$ and less than or equal to $V \times m \times \varepsilon$ are called *valid* area, where $m \in \mathbb{N}$ is a coefficient for calculating the detection period of sensors. Obstacles outside this area cannot enter the safety-critical area within the current detection period, so they should be excluded from the verification in this period.

Collision-avoidance algorithms can turn the AV to any angle, so any heading of the dynamic obstacles can be dangerous. Hence, the initial value of heading is within $\pi$ to $-\pi$ and cannot be reduced, and same for the linear velocity.

**Phased Verification.** Another way of handling large state spaces is to split the verification into several phases, and in each phase, the state space is constrained under a solvable level. For example, when the traveling time of AV is long, the entire journey can be split into multiple sections. As long as the concatenating states between consecutive phases are unchanged, the logic conjunction of verification results of each phase implies the result throughout the entire verification.

## 13.5 Experimental Evaluation

The experiments are conducted on a server with Ubuntu 18.04, 48 CPU, and 256 GB memory. The verification is executed in UPPAAL 4.1.20-stratego-7[4] [9].

[4]The models and external library: https://github.com/rgu01/FM2021.

### 13.5.1 The Collision-Avoidance Algorithm to be Verified

In the following experiments, we employ a state-of-the-art algorithm to demonstrate the ability of our verification approach. The algorithm is based on dipole flow fields [4], and calculates static flow fields for all objects in the map, and dynamic dipole fields for moving objects. When the AV starts to move, the static flow fields generate attractive forces along the reference path to draw the AV to move towards the closest waypoint. When it encounters a dynamic obstacle, dipole fields are generated dynamically and centered by these two moving objects. Magnetic moments are thus calculated in these dipole fields, which push the moving objects away from each other. Therefore, the AV could possibly deviate from its planned path when meeting dynamic obstacles, and thus, it might encounter some static obstacles that are not taken into account by the reference path. Static flow fields now generate repulsive forces surrounding these static obstacles and push the AV away from them. Formulas for calculating these fields and forces can be found in the literature [4]. This algorithm has not been comprehensively verified considering all possible scenarios of dynamic obstacles.

### 13.5.2 Verification Results

In this study, we verify the model containing a C-code library that implements this algorithm, by using our approach. We demonstrate how to find the potential problems of this newly-designed algorithm by using counter-examples returned from the approach, followed by verifying iteratively the improved version.

**Experiment Design.** We report in Table 13.1 several statistics relevant to the obtained results. For each scenario S, we vary the following aspects relevant in real scenarios: (i) WP representing the number of waypoints, (ii) TT that stands for the travelling time of AV, (iii) DO, the number of dynamic obstacles, and (iv) VA, the number of allowed velocities of dynamic obstacles. In scenarios S1 and S2, we use one phase of verification and one allowed velocity, which means that the dynamic obstacle can appear at any moment, always moving at the highest speed throughout the verification. S3 is similar to S1 but it prolongs the travelling time of the AV, and thus, the verification is split into three phases (S3.1 - S3.3). In S4, the dynamic obstacle has three possible

Table 13.1: Verification results of the improved version of the algorithm.

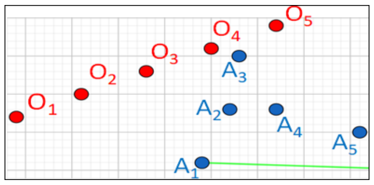
| S | Environment | | Obstacles | | Avoiding Obstacles | | | Reaching Destination | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | WP | TT | DO | VA | NOS | CT | Result | NOS | CT | Result |
| S1 | 2 | 25 | 1 | 1 | 547,617 | 2.7 s | true | 545,505 | 5.5 s | true |
| S2 | 6 | 25 | 1 | 1 | 411,747 | 1.8 s | true | 411,168 | 3.6 s | true |
| S3 | 2 | 85 | 1 | 1 | 3,222,290 | 15.3 s | true | 3,217,767 | 31.8 | true |
| S3.1 | 1 | 30 | 1 | 1 | 1,532,082 | 7.4 s | true | 1,527,811 | 15.7 s | true |
| S3.2 | 1 | 30 | 1 | 1 | 1,183,792 | 5.5 s | true | 1,185,550 | 11.4 s | true |
| S3.3 | 1 | 25 | 1 | 1 | 506,416 | 2.4 s | true | 504,406 | 4.7 s | true |
| S4 | 2 | 15 | 1 | 3 | 12,317,809 | 1.0 mins | true | 12,498,924 | 2.1 mins | true |
| S5 | 2 | 15 | 2 | 1 | 1,398,011 | 7.6 s | false | 226,896,902 | 43.2 mins | true |

velocities, which means its velocity has three initial values and changes arbitrarily during the verification. S5 increases the number of dynamic obstacles to 2, which means there could be at most 2 dynamic obstacles in the map at the same time. For each scenario S, we report the number of states (NOS) and the computation time (CT) needed to verify two requirements, namely obstacle avoiding and destination reaching (see Section 13.4.2 for details). These two values are useful indicators of our approach's performance dealing with various scenarios. All the dynamic obstacles are detected only when they get close to the AV, i.e., they are not foreknown by the AV.

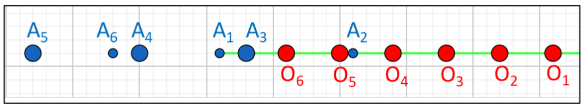
**Problems Discovered by Counter-examples.** Initially, the proposed collision-avoidance algorithm could not pass the reach-avoid verification in any of these scenarios, and we have discovered several problematic scenarios by analyzing the counter-examples returned from our approach:

**Problematic scenario 1.** When there is only one dynamic obstacle whose maximum velocity is less than the maximum velocity of AV, the dipole flow fields generated by the algorithm sometimes draw the AV to the obstacle instead of pushing the obstacle away from it, until their distance is too short (see Fig. 13.8a). This happens because the magnetic moments could push or draw the moving objects. Here, we improve the algorithm by simply turning the direction of the magnetic moments before the AV and the dynamic obstacle get too close.

**Problematic scenario 2.** When the dynamic obstacle and AV move directly towards each other, the dipole fields can only generate magnetic moments on the line of their moving directions, which drive the AV to its opposite direction but on the same line. When the dynamic obstacle keeps moving towards the same direction, the AV can only move back-



(a) Problematic scenario 1



(b) Problematic scenario 2

Figure 13.8: Problematic scenarios discovered by counter-examples. AV's discretized trajectory is blue dots. The dynamic obstacle's discretized trajectory is red dots. AV's reference path is the green line. For differentiation, positions that are too close but belong to different time points are represented by small and large dots in scenario 2. $A_n$ and $O_m$ indicate the AV and obstacle, respectively, $n$ and $m$ are time points.

wards until its distance is longer than a certain value and turns 180° towards its next waypoint, which soon lets the AV get close to the dynamic obstacle again and turn backward (see Fig. 13.8b). According to the counter-examples, this scenario keeps happening iteratively until the AV stops at the boundary of the map, and is hit by the dynamic obstacle eventually. This is the so-called "livelock" scenario that was also discovered by Gu et al. [17]. To overcome this, we force the AV to turn slightly when its heading is opposite to a dynamic obstacle's heading.

**Experimental Results.** Although the improved algorithm passes the verification in S1-S4, our results suggest that it still cannot satisfy the obstacle-avoiding requirement in the last scenarios (S5) that contain more than one dynamic obstacle (see Table 13.1). Note that the destination-reaching property is still satisfied in S5, because the vehicle models are not designed to stop when a collision happens. The rationale of this design is that collisions do not necessarily stop a car from continuing moving. We want to see if the dipole-flow field algorithm can draw the vehicle to its destination anyway when it deviates from the planned paths. Counter-examples are found relatively fast in S5, even though it is more complicated than other scenarios. We leave the further improvement of the algorithm to deal with multiple agents as a future work. The experiments have demonstrated the approach's ability of discovering problems in the early stage of designing collision-avoidance algorithms, and proving the absence of errors in some scenarios for the improved version of the algorithm.

## 13.6 Related Work

Mitsch et al. [18] propose a method to verify safety properties of robots. Their method is based on hybrid system models and differential dynamic logic for theorem proving in KeYMaera. Abhishek et al. [19, 20] also use KeYMaera for collision-avoidance verification. Their models consider the realistic geometrical shapes of vehicles, as well as the combination of maneuvers and braking. Heß et al. [21] propose a method to verify an autonomous robotic system during its operation, in order to cope with changing environments. Our work differs from the above studies in the following aspects: we prove that the reach-avoid verification of nonlinear vehicle models can be simplified to a decidable problem of verifying discrete-time models. In addition, our approach provides counter-examples that are useful to improve the algorithms.

Shokri-Manninen et al. [22] have proposed maritime games as a special case of Stochastic Priced Timed Games and modelled the autonomous navigation using UPPAAL STRATEGO. Their models do not consider the nonlinear kinematics of the vessels, and the options of maneuvers for collision-avoidance are limited. O'Kelly et al. [23] have developed a verification tool, called APEX, and have investigated the combined action of a behavioral planner and state lattice-based motion planner to guarantee a safe vehicle trajectory. In contrast, our approach provides users a generic interface to verify their specific vehicle models equipped with their own collision-avoidance functions. This feature is beneficial to finding bugs in the early stage of designing new algorithms, or employing modified ones.

Although our work relies on the theorems proposed by Fan et al. [8], our work is orthogonal to theirs, that is, their work can be used for the initial construction of reference paths that avoid static obstacles, and our method can be used to verify the dynamic collision-avoidance function of moving obstacles.

## 13.7 Conclusions and Future Work

In this paper, we propose a verification approach to formally verify reach-avoid requirements of autonomous vehicles, assuming nonlinear trajectories of movement. We overcome the difficulty of verifying nonlinear hybrid vehicle trajectories by transforming the latter into discrete-time trajectories whose verification we prove sufficient to guarantee meeting the requirements of the original nonlinear ones. Moreover, we engage tool support (i.e., UPPAAL STRATEGO) that provides users a generic interface to configure and verify their own vehicle models equipped with different collision-avoidance algorithms. We show the abilities of our verification method by model checking a state-of-the-art collision-avoidance algorithm based on dipole flow fields, which discovers bugs not detectable by simulation or testing.

Some interesting directions of future work include: (i) exploring ways of handling complex vehicle models that represent more detailed kinematic features, and (ii) statistical verification of the cases where the distances between dynamic obstacles and AV are smaller than the tracking-error boundaries but collisions do not necessarily occur.

## Acknowledgments

We acknowledge the support of the Swedish Knowledge Foundation via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr: 20150022, and via the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190-038.

# Bibliography

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

[2] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Computer Science Dept., Iowa State University, 10 1998.

[3] 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.

[4] 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.

[5] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, 1997.

[6] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of computer and system sciences*, 57(1):94–124, 1998.

[7] Gerardo Lafferriere, George J Pappas, and Sergio Yovine. A new class of decidable hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 137–151. Springer, 1999.

[8] Chuchu Fan, Kristina Miller, and Sayan Mitra. Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In *International Conference on Computer Aided Verification*, pages 629–652. Springer, 2020.

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

[10] Jonathan A DeCastro, Javier Alonso-Mora, Vasumathi Raman, Daniela Rus, and Hadas Kress-Gazit. Collision-free reactive mission and motion planning for multi-robot systems. In *Robotics research*, pages 459–476. Springer, 2018.

[11] Chuchu Fan, Zengyi Qin, Umang Mathur, Qiang Ning, Sayan Mitra, and Mahesh Viswanathan. Controller synthesis for linear system with reach-avoid specifications. *IEEE Transactions on Automatic Control*, 2021.

[12] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[13] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. In *International journal on software tools for technology transfer*, pages 134–152. Springer, 1997.

[14] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking in Dense Real-Time. *Information and computation*, 104(1):2–34, 1993.

[15] André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *Journal of Logic and Computation*, 20(1):309–352, 2010.

[16] Rong Gu, Cristina Seceleanu, Eduard Paul Enoiu, and Kristina Lundqvist. Formal verification of collision avoidance for nonlinear autonomous vehicle models. Technical report, Mälardalen University, April 2021.

[17] 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.

[18] Stefan Mitsch, Khalil Ghorbal, David Vogelbacher, and André Platzer. Formal verification of obstacle avoidance and navigation of ground robots. *The International Journal of Robotics Research*, 36(12):1312–1340, 2017.

[19] Aakash Abhishek, Harry Sood, and Jean-Baptiste Jeannin. Formal verification of braking while swerving in automobiles. In *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2020.

[20] Aakash Abhishek, Harry Sood, and Jean-Baptiste Jeannin. Formal verification of swerving maneuvers for car collision avoidance. In *2020 American Control Conference (ACC)*, pages 4729–4736. IEEE, 2020.

[21] Daniel Heß, Matthias Althoff, and Thomas Sattel. Formal verification of maneuver automata for parameterized motion primitives. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1474–1481. IEEE, 2014.

[22] Fatima Shokri-Manninen, Jüri Vain, and Marina Waldén. Formal verification of colreg-based navigation of maritime autonomous systems. In *International Conference on Software Engineering and Formal Methods*, pages 41–59. Springer, 2020.

[23] Matthew O'Kelly, Houssam Abbas, Sicun Gao, Shin'ichi Shiraishi, Shinpei Kato, and Rahul Mangharam. Apex: Autonomous vehicle plan verification and execution. *SAE World Congress*, 2016.

# Appendix A

## A.1 Algorithm of Synthesis

Algorithm 8 is the simplified pseudo-code of running Query (12.3) in UPPAAL STRATEGO. Details of this algorithm are in the literature [26].

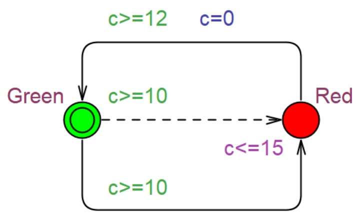
## A.2 Algorithm of Verification and Labeling

In Algorithm 6, line 3 passes the initial state $S_0$ of the TG $\mathcal{G}$ and the negation of the state formula of Query (12.6), i.e., $\neg\phi$, to the function `Delay`, which adds the symbolic succeeding states of $S_0$ via restricted delay transitions. The definition of restricted delay transitions is presented in literature [27]. In this paper, we adapt this function on symbolic states (i.e., zones) by using difference bounded matrices (DBM) in UPPAAL. Fig. A.1 shows an example of a UTA modeling a traffic light and its symbolic semantic model - a Zone Graph. The action transitions and delay transitions are arrows labeled with `a` and `d`, respectively. An example of symbolic states that are used in the `Delay` function is <`Red, c=0`> in Fig. A.1b. Briefly, if the action is the only controllable action at state $S$, the function `Allow` returns *true* directly, which is the case at the initial state in Fig. A.1b; otherwise, it looks up the strategy and finds the set of the best actions that have the highest score at the current state (i.e., $best(\sigma, S)$). If the current action belongs to the set, it is allowed and we call the *label* function to label the state-action pair as *visited* (line 31).

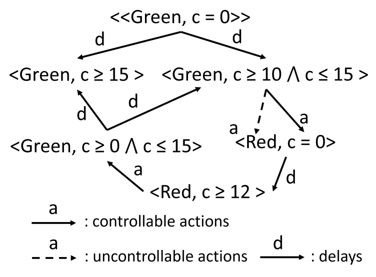
When the delay transition is allowed in the function `Delay`, we continue to check if the succeeding state $S'$ is not in the stack $SD$ and

**Algorithm 8:** Simplified algorithm behind the `minE`-query (adapted from Algorithm 1 in the literature [26])

```
minE(tg, iterationNum, totalNum, goodNum, formula)
int $iterations = 0$
int $bestFitness = \infty$
Strategy $best = empty$
Strategy $aStrategy = empty$
for $iterations < iterationNum$ do
    int $totalRuns = 0$
    int $goodRuns = 0$
    for $totalRuns < totalNum$ do
        Run $aRun$ = simulate($tg$, $aStrategy$)
        if $aRun$ satisfies formula then
            $aStrategy$ = learn($aRun$)
            $goodRuns{+}{+}$
            if $goodRuns \geq goodNum$ then
                break
        $totalRuns{+}{+}$;
    if $goodRuns \geq goodNum$ then
        $fitness$ = evaluate($aStrategy$)
        if $fitness < bestFitness$ then
            $bestFitness = fitness$
            $best = aStrategy$
    $iterations{+}{+}$
return $best$;
```



(a) An example of TG



(b) The Zone Graph of the TG

Figure A.1: An example of a TG and its semantic model.

satisfies the invariant at the location of the current state ($I(S.l)$) and the restriction ($\varphi$) (line 10). The restriction $\varphi$ is actually $\neg\phi$, which means the state space exploration only visits the states where the state formula $\phi$ of Query (12.6) is *false*, because the verification of a liveness property aims to find a run where $\phi$ is *false* at all states as the counter-example. If $S'$ satisfies the condition (line 10), it is pushed into the stack $SD$ for further exploration. In Fig. A.1b, after delaying at the initial location, two symbolic states can be reached, which are passed to function `Search` as the value of parameter $S$. The restriction $\neg\phi$ is also passed to function `Search` as the value of parameter $\varphi$.

In function `Search`, we first check if the current state $S$ satisfies $\varphi$ (line 13, which returns an empty state when $\varphi$ is *false* at $S$, and $S$ itself when $\varphi$ is *true*). At line 15, the function checks if there is a loop in the state space by checking if the current state $S$ is in the stack $ST$. If a loop exists, an unsatisfactory run (the runs where no state satisfies $\phi$) is found and thus the algorithm exists with a negative result of verification; otherwise, we check if the maximum run is found (line 17). According to the definition in the literature [27], a run is maximal if either it ends in a state with no outgoing transitions, ends in a state from which an unbounded delay is possible, or is infinite. When such runs are found, no further symbolic state exists and thus the algorithm exists with a negative result of verification; otherwise, the algorithm pushes $S$ into $ST$ and continues to explore the unvisited states (line 20). For example, in Fig. A.1b, both succeeding states of the initial state are pushed into $SD$ and explored by function `Search`. The state `<Green, c≥15>` ends at a deadlock, whereas the state `<Green, c≥10 ∧ c≤15>` has two actions, that is, a controllable action and an uncontrollable one. Both actions end to the same state `<Reg, c=0>`.

Similar to the function `Delay`, line 22 explores the succeeding states via controllable actions that are allowed by the strategy $\sigma$, or uncontrollable actions. If a controllable action is allowed, its succeeding states are recursively explored at line 25. For example, at the state `<Green, c≥10 ∧ c≤15>` in Fig. A.1b, we can either choose the uncontrollable action without asking the strategy, or choose the controllable action after asking the strategy, and then continue to explore the state space in the same manner.

Assume we instantiate a model of the TG in Fig. A.1a, namely `trafficLight`, and we want to verify a liveness property: `A<> trafficLight.Red`, by following algorithm 6, we will get a negative result of verification with a counter-example returned, that is, a trace from the initial state «Green, c=0> to the state <Green, c≥15>.

## A.3 Templates of the TG models

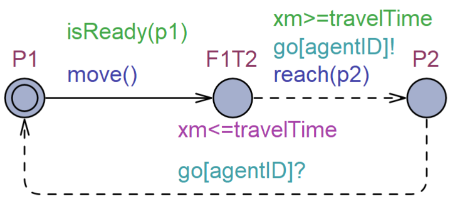


Figure A.2: The TG template of agent movement

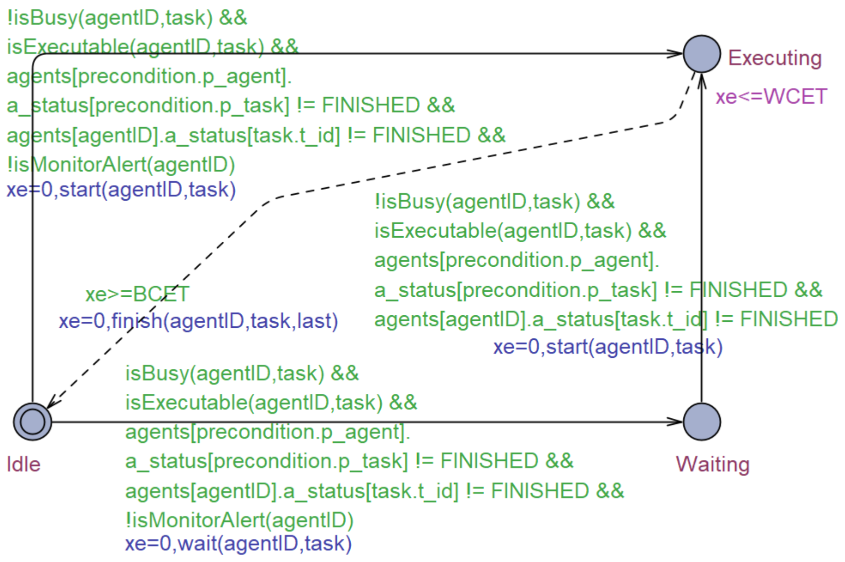

!isBusy(agentID,task) &&
isExecutable(agentID,task) &&
agents[precondition.p_agent].
a_status[precondition.p_task] != FINISHED &&
agents[agentID].a_status[task.t_id] != FINISHED &&
!isMonitorAlert(agentID)
xe=0,start(agentID,task)
Executing
xe<=WCET
!isBusy(agentID,task) &&
isExecutable(agentID,task) &&
agents[precondition.p_agent].
a_status[precondition.p_task] != FINISHED &&
agents[agentID].a_status[task.t_id] != FINISHED
xe=0,start(agentID,task)
xe>=BCET
xe=0,finish(agentID,task,last)
isBusy(agentID,task) &&
isExecutable(agentID,task) &&
agents[precondition.p_agent].
a_status[precondition.p_task] != FINISHED &&
agents[agentID].a_status[task.t_id] != FINISHED &&
!isMonitorAlert(agentID)
xe=0,wait(agentID,task)
Idle
Waiting

Figure A.3: A TG template of agent task execution

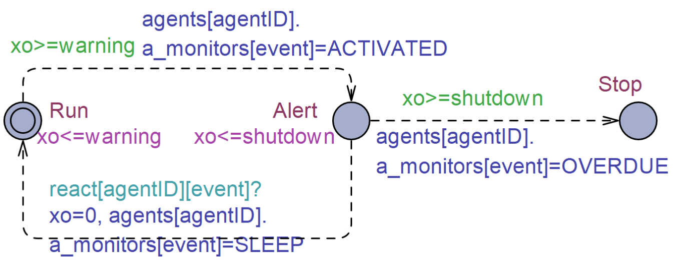


Figure A.4: The TG template of agent monitors

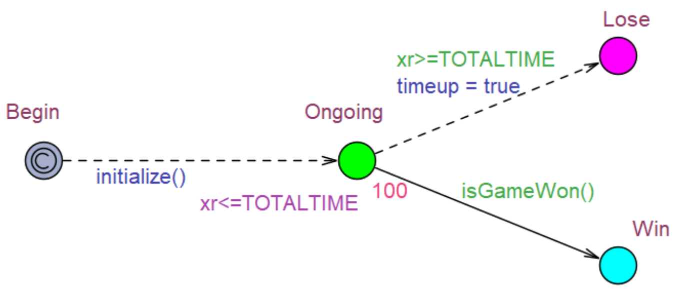


Figure A.5: The Referee TG

## A.4 Overview of the External Library of MoCReL

The new extension of UPPAAL STRATEGO supports calling external libraries that are implemented by C/C++. An example of the implementation is in: `https://github.com/DEIS-Tools/stratego$_$external$_$learning`. The library must contain the following functions so that UPPAAL STRATEGO can invoke it correctly:

```c
// Allocates an instance of a learner
void* uppaal_external_learner_alloc(bool minimization, size_t
    d_size, size_t c_size, size_t a_size);
// Deallocation code for object
void uppaal_external_learner_dealloc(void* object);
// print out strategies
char* uppaal_external_learner_print(void* object);
// Deep-copy function of an instance of a leaner
void* uppaal_external_learner_clone(void* object);
// Called for each sample in a trace
void uppaal_external_learner_sample_handler(void* object, size_t
    action, double* from_d_vars, double* from_c_vars, double*
    t_d_vars, double* t_c_vars, double value);
// Return the values of state-action pairs in the strategy
double uppaal_external_learner_predict(void* object, bool
    is_search, size_t action, double* d_vars, double* c_vars);
// Batch-completion call-back
void uppaal_external_learner_flush(void* object);
```

When running MoCReL in UPPAAL STRATEGO, the function `alloc` is firstly called, which instantiates the learner. Next, when Query (12.3) is executed, UPPAAL STRATEGO simulates the model to sample runs, which are passed to the learner by calling the function `sample_handler`. During the simulation and verification, wherever the model has more than one controllable actions, function `predict` is called for looking up the strategy and returning the value of the action at the current state. This value can be used as the probability or the weight of choosing that action, which is introduced in Subsection 12.4.4. Additionally, when under verification (Query (12.6) is being executed), MoCReL marks the chosen state-action pairs in the function `predict` so that the strategies can be compressed after the verification passes. One can print the strategy by using a query `saveStrategy(path)` in UPPAAL STRATEGO. It will call the function `print` to print the strategy to the specific file in a standard format.

# Errata

This errata sheet lists errors and their corrections for the doctoral dissertation of Rong Gu, titled "Formal Methods for Scalable Synthesis and Verification of Autonomous Systems - Mission Planning and Collision Avoidance", Mälardalen University, 2022.
ISBN: 978-91-7485-552-4.

- Location: Page 155

  Error: Completeness in Table 9.4

  Correction: Completeness in Table 9.4 is strictly connected to the satisfaction of liveness queries in the form of `A<> P under opt`, and not the method's completeness meaning that whenever a valid strategy exists, the algorithm must find it, in which sense MCRL is not complete.

- Location: Page 287

  Error: Example of two runs. One exists in both non-deterministic strategies and stochastic strategies but one exists only in non-deterministic strategies.

  Correction: Both runs in the example can exist in non-deterministic strategies and stochastic strategies. However, nondeterministic choices assume "indifferent" probability, which means any run is possible. We remove the example but keep the conclusion of inclusion relation reached by Theorem 1 at Page 286.