View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Chalmers Research

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Correct-by-Construction Tactical Planners for Automated Cars

Jonas Krook

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Göteborg, Sweden, 2019

Correct-by-Construction Tactical Planners for Automated Cars

Jonas Krook

Copyright © 2019 Jonas Krook
All rights reserved.

ISSN 1403-266X
This thesis has been prepared using LaTeX.

Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Phone: +46 (0)31 772 1000
www.chalmers.se

Printed by Chalmers Reproservice
Göteborg, Sweden, December 2019

Abstract

One goal of developing automated cars is to completely free people from driving tasks. Automated cars that require no human driver need to handle all traffic situations that a human driver is expected to handle, and possibly more. Although human drivers cause a lot of traffic accidents, they still have a very low accident and failure rate that automated systems must match.

Tactical planners are responsible for making discrete decisions during the coming seconds or minute. As with all subsystems in an automated car, these planners need to be supported with a credible and convincing argument of their correctness. The planners' decisions affect the environment and the planners need to interact with other road users in a feedback loop, so the correctness of the planners depend on their behavior in relation to other drivers and the environment over time. One possibility to ascertain their correctness is to deploy the planners in real traffic. To be sufficiently certain that a tactical planner is safe by that methods, it needs to be tested on 255 million miles without having an accident.

Formal methods can, in contrast to testing, mathematically prove that the requirements are fulfilled. Hence, they are a promising alternative for making credible arguments of tactical planners' correctness. The topic of this thesis is how formal methods can be used in the automotive industry to design safe tactical planners. What is interesting is both how automotive systems should be modeled in formal frameworks, and how formal methods can be used practically within the automotive development process.

The main findings of this thesis are that it is natural to express desired properties of tactical planners in formal languages and use formal methods to prove their correctness. Model Checking, Reactive Synthesis, and Supervisory Control Theory have been used in the design and development process of tactical planners, and all three methods have their benefits, depending on the application.

Formal synthesis is an especially interesting class of formal methods because they can automatically generate a planner based on requirements and models. Formal synthesis removes the need to manually develop and implement the planner, so the development efforts can be directed to formalizing good requirements on the planner and good assumptions on the environment. However, formal synthesis has two limitations: the resulting planner is a black box that is difficult to inspect, and it is difficult to find a level of abstraction that allows detailed requirements and generic planners.

Keywords: Formal methods, automated cars, tactical planning, formal verification, formal synthesis, Model Checking, Reactive Synthesis, Supervisory Control Theory.

List of Publications

This thesis is based on the following publications:

Paper A **Jonas Krook**, Anton Zita, Roozbeh Kianfar, Sahar Mohajerani, and Martin Fabian. "Modeling and Synthesis of the Lane Change Function of an Autonomous Vehicle", *IFAC-PapersOnLine, 14th IFAC Workshop on Discrete Event Systems (WODES)*, 2018.

Paper B **Jonas Krook**, Lars Svensson, Yuchao Li, Lei Feng, and Martin Fabian. "Design and Formal Verification of a Safe Stop Supervisor for an Automated Vehicle", *2019 International Conference on Robotics and Automation (ICRA)*, 2019.

Paper C Zahra Ramezani, **Jonas Krook**, Zhennan Fei, Martin Fabian, and Knut Åkesson. "Comparative Case Studies of Reactive Synthesis and Supervisory Control", *2019 18th European Control Conference (ECC)*, 2019.

Paper D **Jonas Krook**, Roozbeh Kianfar, and Martin Fabian. "Formal Synthesis of Safe Stop Tactical Planners for an Automated Vehicle", Submitted to *Workshop on Discrete Event Systems (WODES)*, 2020.

Acknowledgments

I would like to thank Mattias Bengtsson for encouraging me to pursue a Ph.D. I had my doubts back then, but since I started my Ph.D. studies I have never felt that I made a bad choice; not even when I was writing this thesis.

Thanks also to Martin Fabian and Roozbeh Kianfar who have supported me with feedback, good discussions, and direction, without which I would have been completely lost.

Thank you very much Emilia for being so loyal and for supporting me at all times. I would never have gotten this far without you.

Jonas Krook
Göteborg, December 2019

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Acronyms

AD:	Automated Driving
ADAS:	Advanced Driver Assistance Systems
ASIL:	Automotive Safety Integrity Level
BDD:	Binary Decision Diagram
DES:	Discrete Event System
DL:	Deep Learning
E/E:	Electronics or Electrical
EFSM:	Extended Finite State Machine
FSM:	Finite State Machine
GR(1):	General Reactivity of Rank 1
LSM:	Lateral State Manager
LTL:	Linear Temporal Logic
MC:	Model Checking
ML:	Machine Learning
RS:	Reactive Synthesis
SCT:	Supervisory Control Theory

Contents

Abstract **i**

List of Papers **iii**

Acknowledgements **v**

Acronyms **vi**

I Overview **1**

1 Introduction **3**
1.1 Hypothesis 8
1.2 Research questions 8
1.3 Contributions 8
1.4 Method 9
1.5 Outline 9

2 How safe? **11**
2.1 Driving Around the World 11
2.2 ISO26262 12
2.3 Saving Some Time 14

3 Formal Methods **15**
3.1 Automata 15
3.2 LTL 16
3.3 Terminology 17

3.4 Model Checking 18
3.5 Reactive Synthesis 19
3.6 Supervisory Control Theory 21

4 Correct-by-Construction Tactical Planners **25**
4.1 Inspection of Results 26
4.2 Verification vs. Synthesis 28
4.3 Abstraction of States 30
4.4 Modeling Considerations 31

5 Conclusions **37**
5.1 Future Works 38

6 Summary of Included Papers **39**

References **41**

II Papers **47**

A Modeling and Synthesis of a Lane Change Function **A1**
1 Introduction A3
2 Preliminaries A5
2.1 Extended Finite-State Machines A5
2.2 Controllability A7
2.3 BDD-based Synthesis A7
2.4 Compositional Synthesis A8
3 System Description and Modeling A8
4 Specification A10
5 Synthesis A11
6 Conclusion A13
References A13

B Formal Verification of a Safe Stop Planner **B1**
1 Introduction B3
2 System architecture B5
3 Supervisor Design B7
3.1 Formal requirements B8
3.2 Modeling in Stateflow B10
3.3 Model checking B12
4 Results and Discussion B12
4.1 Formal Verification B12
4.2 RCV Experiment B13

5 Conclusion and Future Work B15
References B16

C Comparative case studies of RS and SCT **C1**
1 Introduction C3
2 Preliminaries C4
2.1 Reactive Synthesis C4
2.2 Supervisory Control Theory C6
3 Stick-Picking Game C7
3.1 TuLiP C7
3.2 Supremica C10
4 Autonomous Driving C10
4.1 TuLiP C11
4.2 Supremica C13
5 Comparison C15
6 Conclusion C18
References C19

D Formal Synthesis of Safe Stop Planners **D1**
1 Introduction D3
2 Preliminaries D5
2.1 Supervisory Control Theory D5
2.2 Reactive Synthesis D6
3 Scenario D7
3.1 Transport mission D8
3.2 Safe stop D8
3.3 Architecture D9
3.4 Requirements D10
4 Results D10
4.1 Vehicle, trajectory planner, and controller D11
4.2 Requirements D12
4.3 Synthesis Result D14
5 Conclusion D16
References D17

Part I

Overview

CHAPTER 1

Introduction

One goal of developing automated cars is to completely free people from driving tasks, and the introduction of such cars is thought to bring several benefits and new application areas. For instance, as many people spend a sizable part of their time commuting to and from work in their own cars, one major benefit is to let people spend that commuting time on more fun things than driving on congested urban roads. Another example is that automated cars would allow people who currently cannot or should not drive to get access to better personal mobility, for instance elderly people or people with visual impairments [1]–[4].

Automated cars that require no human driver are technologically very advanced machines, and they need to handle all traffic situations that a human driver is expected to handle, and possibly more. However, this level of automation is difficult to achieve [5], so automated cars that still partly rely on an active human driver might be an important step to making traffic safer [6]. Such cars with less advanced automation can be beneficial for drivers in other ways as well, for instance, cars that keep themselves at a certain speed and in lane with driver supervision might be considered a benefit to some drivers.

The amount of human involvement in the driving tasks is the main factor that affects the difficulty to implement an automated vehicle, so there is a classification system which categorizes vehicles into one of 6 levels depending on required driver involvement (see Table 1.1) [7]:

Level 0 The car does not help with driving, so the driver performs all tasks related to controlling speed, steering, etc.

Level 1 The car steers or controls speed; adaptive cruise control is an example.

Level 2 The car steers and controls speed, but the driver needs to continuously monitor it; cars that stay in lane or parks themselves are examples.

Level 3 The car steers and controls speed, but the driver might need to take over control in difficult situations; cars in this level might for instance overtake on their own.

Level 4 The car completely manages the driving tasks within certain operational conditions; a car that can only drive on dry highways without human intervention would be in this level.

Level 5 The car completely manages the driving tasks; there need not be anyone in the car.

Related to the automation levels are Advanced Driver Assistance Systems (ADAS) and Automated Driving (AD) systems. In this thesis, ADAS refer to functionality that help the driver with the driving tasks in Level 0 to 2, and AD systems refer to functionality that perform the driving task in Level 3 to 5.

SAE Level	Name	Monitoring agent	
Level 0	No automation	Driver	
Level 1	Driver assistance	Driver	Hands on
Level 2	Partial automation	Driver	Hands off
Level 3	Conditional automation	System	Eyes off
Level 4	High automation	System	Mind off
Level 5	Full automation	System	Steering wheel optional

Table 1.1: SAE automation levels [7].

Reasons for both ADAS and AD are to relieve the driver from driving tasks and to increase traffic safety. Some ADAS, like adaptive cruise control, are supposed to free the driver from controlling speed and distance, and instead let the driver focus on planning ahead, and in that way facilitate safer driving. Other ADAS correct for errors, of which autonomous emergency braking is an example; it overrides the driver's commands and automatically brakes the vehicle if the system believes that the driver cannot avoid a collision. The AD systems are supposed to increase traffic safety by removing the driver altogether.

Although human drivers cause a lot of traffic accidents, they still have a very low accident and failure rate [8]. This low rate poses difficulties when both ADAS and AD systems are developed; if these systems shall increase traffic safety, then their introduction cannot cause higher risk of accidents than what is expected without them. Hence, the low accident rate of human drivers effectively puts requirements on how fault tolerant the ADAS and AD systems need to be. However, this does not mean that all ADAS

and AD systems have the same requirements on fault tolerance. Systems in higher SAE levels generally require higher fault tolerance. For instance, a collision warning system in Level 0 can easily be ignored by the driver and a high false warning rate can therefore be tolerated since it does not affect the safety significantly. The occurrence of erroneous and excessive braking in a Level 3 system, on the other hand, must be rare lest the vehicle runs an undue risk of being rear-ended. A more detailed motivation for this is given later in Section 2.2.

When human driving tasks are taken over by automation it is done so by several electronics or electrical (E/E) subsystems, each with a specific sub-task. For the purpose of this thesis the automated car's E/E subsystems are broken up into four categories according to the overview shown in Figure 1.1. Starting from the left, the sensors perceive the vehicle's surroundings and turn that into sensor-specific semantic information. For instance a camera can find other vehicles in an image. The information from each separate sensor is then passed on to the sensor fusion subsystem. It takes in information from all the sensors and merges it to produce a more accurate estimation of the environment. Based on the fused sensor information, decision and control decides which actions to take and how. Decision and control passes on its decisions to the actuators in form of, e.g., paths and acceleration commands. The actuators convert the decisions to, e.g., steering torque and brake torques.

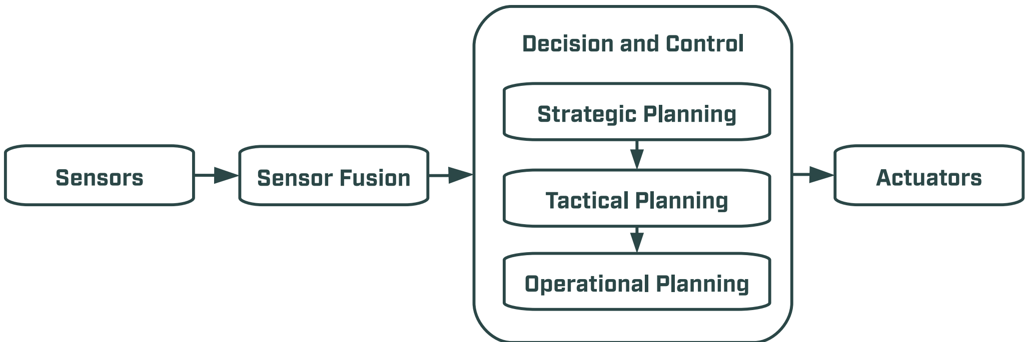

Figure 1.1: System overview.

All four subsystems can be further divided, but here it is only interesting to consider decision and control which can be divided into strategic, tactical, and operational planning [9], as can be seen in Figure 1.1. Strategic planning refers to planning for long time horizons, such as route planning. Tactical planning refers to planning during the coming seconds or minute, which would include discrete decisions such as when to do lane changes or whether to stop at an intersection. The operational planning is performed on the scale of milliseconds up to a second, and examples can be speed control and sudden avoidance maneuvers.

All four subsystems in Figure 1.1 need to be supported by a credible and convincing

argument of their correctness [10]. Since the subsystems have different tasks and solve them with widely different algorithms their correctness are also ascertained with different methods. The actuators and the operational planning subsystems are analyzed and verified with methods from control theory, which provides tools for asserting stability, and determining maximal tracking error, etc. This is a field with mature analytical methods developed from the 1930's and onward. The sensors and sensor fusion subsystems can use a statistical approach to verify its correctness in open-loop simulations. Given a recorded input an accuracy of the estimates compared to ground-truth data can be calculated. The main issue for AD systems is that a lot of data is needed [8], and that reliable ground-truth data or annotations can be laborious to attain.

Many of the analytical tools from control theory are not suitable for discrete decisions made by the tactical planners that are the main concern of this thesis. For example, analytical proofs of stability and reachability for continuous or discrete time systems with continuous states cannot be used on discrete state systems. The open-loop simulations that are useful for the sensing parts can also not be used to the same extent for the tactical planning subsystem. This so since the planner's decisions affect the environment and the planner needs to interact with other road users in a feedback loop. A recording of a traffic scenario will therefore only be a valid basis for a convincing correctness argument if the planners take the exact same decisions as the vehicle did during the recording; an implausible feat for the millions of miles needed as indicated by Kalra and Paddock [8]. Instead of simulations, the tactical planners can be verified as part of the complete vehicle during real driving. However, this does not scale well since failures during the verification will add millions of kilometers to the needed total mileage [10]. This thesis explores ways to limit the needed verification effort of tactical planners, and tries to focus on the correctness of their construction to achieve fault tolerant planners.

The above gives some indication on how much effort that is needed to verify that a functionality is safe enough from a complete vehicle level. However, needed safety effort must be known already during the development of the E/E subsystems. It can be difficult to reason about and argue for how fault tolerant these individual ADAS or AD E/E subsystems need to be, so the automotive industry has developed the functional safety standard ISO26262 [11] to establish best practices to deal with the risks introduced by safety-critical functionality. In broad strokes, the standard defines a systematic process to judge the safety criticality of different end effects that can be caused by different E/E system faults. The safety criticality measure has 5 different levels and is called Automotive Safety Integrity Level (ASIL). Based on the ASIL for an end effect, the standard then recommends best practice methods to use to ensure a sufficient level of safety when designing, implementing, verifying, and validating parts of the E/E-system that can cause that end effect if they experience a failure. For instance, one possible end effect of a fault could be that the vehicle does not accelerate when the driver presses the accelerator pedal, and the fault might be a sensor that fails to measure the position of the accelerator pedal correctly. A vehicle that cannot accelerate has a problem, but

maybe not a safety critical one, and in that case the ASIL would be low. A low ASIL would in turn mean that the accelerator pedal sensor could be developed with less safety effort and still keep the risk of harm at a sufficiently low level.

ISO26262 recommends two verification methods: *reviews* and *testing*. Reviews are qualitative and requires staff to go through and inspect work products. Testing is quantitative and measures the effect when systems are exposed to certain inputs. Both these verification methods can be applied to large parts of the development process [11], but neither of them can provide proofs of correctness; they can only find faults, not an absence of them [12]. Also, when the design or implementation of a part is changed the review and testing has to be performed again, and with higher ASIL the reviews and testing are recommended to be more thorough. This might be a bottleneck if more advanced ADAS and AD systems are to be developed. More advanced could mean several things, but two examples could be more capabilities or less driver involvement. Both these advances incur more work spent on reviews and testing, which is discussed more in Section 2.2.

The ISO26262 standard mentions one class of methods and tools that can alleviate some burdens of reviews and testing. These are called *formal methods* [12], [13], and can, in contrast to reviews and testing, mathematically prove that the requirements are fulfilled. (A caveat being that this statement holds for a model of the system, so any modeling errors void the correctness proof.) At their core, formal methods have a formal language that has well defined and unambiguous semantics. The formal language is used to formalize the requirements. After the requirements are formalized there are mainly two subclasses of formal methods from which to choose to continue the process; formal verification and formal synthesis.

Formal verification needs a formal model of the implementation and possibly the environment with which it interacts [12]. This model can be the actual software code, a simplified abstraction thereof, or a very simple model of the intended functionality of the item; it depends on which requirements are being verified against. When the formal requirements and the formal model are available the formal verification tool can search for counterexamples. If none are found, then the model fulfills the requirements. Because formal verification comes with a proof or a counterexample, less effort needs to be spent on reviews and tests.

Formal synthesis, on the other hand, does not have a model of the implementation included in the formal model, but it requires a model of the environment [13], [14]. Based on the formal requirements and the formal model, formal synthesis automatically creates a correct-by-construction implementation. This has the advantage over formal verification that the implementation does not need to be performed manually; an error prone and time consuming process. Also, changes in the requirements does not necessarily incur any manual reimplementation.

Formal methods, especially formal verification, have been used successfully in industrial applications [15]–[17]. They have a clear potential also in the automotive industry to

increase the quality of the safety work, as well as decrease development time and cost. However, except in the case of software verification, it is not established in any large extent as to what parts of the automotive development process that can benefit from formal methods. Potential pitfalls and limitations are also not well explored.

The topic of this thesis is how formal methods can be used in the automotive industry to design safe tactical planners. What is interesting is both how automotive systems should be modeled in formal frameworks, and how formal methods can be used practically within the automotive development process.

1.1 Hypothesis

The hypothesis of this thesis starts in the belief that formal methods, and especially formal synthesis, can be applied in automotive development processes to generate useful and meaningful correct-by-construction tactical planners. Such formal methods should be beneficial because they alleviate drawbacks of other methods that cannot provide correctness proofs, specifically the processes of reviews and testing. However, since formal methods, and especially formal synthesis, does not seem to be widely adopted for generation of tactical planners for automated cars there probably are limitations or obstacles that hinder the adoption. These potential limitations and obstacles are investigated in this thesis.

Furthermore, with discrete systems operating in continuous domains there must be a mapping, or abstraction, from continuous states to discrete states. This abstraction affects how a tactical planner can operate, and selecting the wrong level of abstraction could cause synthesis to generate useless or trivial planners, or fail to generate planners at all. This thesis investigates whether and how the abstraction of the continuous domain affects the usefulness of the tactical planners.

1.2 Research questions

Based on the hypothesis, this thesis aims to answer the following research questions:

- What are the current limitations of formal synthesis that currently hinders industrial adoption for automotive systems?
- What level of abstraction is possible to use for automotive systems, and will that level facilitate meaningful and non-trivial planners?
- To what problems can synthesis and verification be applied?

1.3 Contributions

The main contributions of this thesis are:

- It identifies the difficulty to inspect the result of synthesis as an obstacle to adoption of formal synthesis.
- It identifies a conflict between the desire to express detailed requirements and to generate generic tactical planners.
- It provides insights into how verification and synthesis models differ when it comes to separation of environment, requirements, and planner.
- It determines that the differing syntax and semantics of the modeling paradigms have little effect on the synthesized planners. However, a few characteristics of the synthesis results of the different paradigms are important to consider when formal synthesis is applied to a problem.

1.4 Method

The work of this thesis has been carried out in the form of several case studies connected to tactical planners for automated cars. Realistic requirements and scenarios have been formulated in each case, and formal methods have been used to design correct-by-construction tactical planners. These planners have then been evaluated by simulations, experiments, and visual inspections. The evaluations and comparisons have mainly been qualitative.

1.5 Outline

This thesis is divided into two parts. Part I gives an introduction to safety of automated cars, explains fundamentals of formal methods, and puts the appended papers into context. Part II contains the appended papers that form the basis of this thesis.

Part I consists of the following chapters: Chapter 2 presents the problem of credibly arguing for the safety of automated cars. Chapter 3 goes through the fundamental concepts of formal methods that are used in later chapters. The appended papers are discussed and put into perspective in Chapter 4. Chapter 5 concludes the findings in this thesis and presents future challenges. Finally, Chapter 6 gives short summaries of the appended papers.

CHAPTER 2

How safe?

In order to deploy AD systems on public roads, the automotive industry needs to present compelling arguments that the cars are sufficiently safe [10]. One way of providing such arguments, as mentioned in Chapter 1, is to follow the process recommended by ISO standard 26262 [11]. This in practice means that for certain safety requirements the probability of failure should be less than 10^{-9}. Achieving such level of integrity and correctness makes any approach only based on simulation, field testing and statistical data inadequate and unrealistic. As noted by [8], if a fleet of 100 autonomous vehicles drives 24 hours/day, 365 days/year, at an average speed of 25 mph, it takes 400 years to demonstrate with 95% confidence that their failure rate is 20% better than the human driver failure rate of 1.09 fatalities per 100 million miles (US, 2013). Furthermore, if there is a failure during this verification, the amount of required field testing is increased [10].

This chapter gives an overview of how many miles have been driven by some automated car systems and an introduction to the functional-safety standard ISO 26262.

2.1 Driving Around the World

Several companies working with automated cars are publishing safety reports that detail their efforts to ascertain safety [18]–[21]. They are also publishing how long they have driven with their field test vehicles (or customer vehicles in the case of Tesla): Waymo 10 million miles [22]; Tesla 1 billion miles [23]; Yandex 1 million miles [24]; Uber at least 2 million miles [25], [26]. All of these companies, except Tesla, are field testing SAE Level 4 or 5 cars. Tesla's system is available for consumers and relies on the driver as a

fall-back, so it is at most a SAE Level 3 system (cf. Table 1.1).

Is this amount of driving enough to credibly argue that the systems are safer than a human driver? To be 95 % sure that an automated car's systems are at least as safe as a human driver (in the U.S.) the systems need to drive 255 million miles with no fatalities [10]. As can be seen above, most automated-car companies fall short on this target with at least one order of magnitude. Tesla seems to fare well since they have tested on 4 times as many miles as required. However, fatalities increase the required number of miles; 402 million miles required for one fatality and 535 million miles for a second fatality, etc [10]. And there have been fatalities where, for instance, Tesla and Uber cars have been involved [27], [28]. An independent report also argues that Tesla cars with active AutoPilot are less safe than when AutoPilot is inactive [29].

The numbers above also assumes that the systems have a single occupant [10]. When the cars are put into use one has to consider the expected number of occupants and increase the number of miles driven accordingly.

A further caveat is that the statistical argument of number of miles driven without fatalities must be based on one and the same version of the system, a prerequisite which is clearly void for Tesla [30], but it is also unlikely that the other actors have been driving millions of miles for several years without updating their systems.

With these mileages in mind, it seems infeasible to base a correctness argument solely on the amount of miles driven by an ADAS or AD system. The question is then: what is the alternative?

2.2 ISO26262

As mentioned above and in Chapter 1, it can be difficult to reason about and argue for how fault tolerant individual subsystems of ADAS or AD systems need to be, so the automotive industry has developed the functional safety standard ISO26262 to establish best practices to deal with the risks introduced by safety-critical functionality [11]. Simply put, the process of the standard starts with a definition of the functionality to be developed (called the 'item' in the standard), and then a hazard analysis is performed for the item. During the hazard analysis potential malfunctions of the item are considered, and if they are possible sources of harm they are considered hazards. Each hazard is considered in all the operational situations of the item, and the combination of the hazard and an operational situation is called a hazardous event. The purpose of the standard is to provide best practice recommendations on how to reduce the risk of all hazardous events to an acceptable level[1].

For each hazardous event it is judged how severe the consequences of that event can be (severity), how often the vehicle is in the operational situation of the hazardous event (exposure), and how difficult it is for an agent other than the safety-critical functionality

[1]Note that there is still a non-zero risk for the occurrence of the hazardous event; there is no such thing as 100 % safe.

to detect and counteract the hazardous event (controllability). For instance, one identified possible hazard could be the omission of braking when approaching a stationary vehicle. The severity is decided by the collision speed, and the exposure is decided by the probability of a stationary vehicle on a road where such speed is held. The controllability is based on how likely the driver is to brake, which, among other things, is influenced by the automation level.

The purpose of the standard is to prevent the hazardous events, and the means with which to prevent them are specified in safety goals. To prevent fatal collisions with pedestrians, for instance, a safety goal might be to limit the top speed of the vehicle to 30 km/h. For every identified hazardous event, the estimated severity, exposure, and controllability are combined to form an automotive safety integrity level (ASIL). The ASIL is transfered to the safety goals that are attached to a hazardous event, and the ASIL will guide what safety efforts that are needed during development to ascertain the fulfillment of each safety goal.

When all safety goals have been formulated, the standard follows a V-model: an architecture with subsystems is developed for the item, and the safety goals are broken down into functional safety requirements, allocated to the subsystems. The subsystems are further split into hardware and software components by hardware and software architectures. Correspondingly, the functional safety requirements are further refined into lower level safety requirements allocated to the hardware and software components. All the architectural components (subsystems, hardware and software components) are then implemented, verified, and validated in accordance with the safety requirements. Here, the ASIL of the safety goal has a direct effect on the effort as the standard prescribes recommended implementation, verification, and validation activities based on the ASIL, and, in general, the ASIL of a safety goal is inherited to all the lower level requirements that refines it.

From a functional safety point of view, ADAS and AD systems are increasingly difficult to develop because of three mechanisms. Firstly, as desired operational situations and capabilities of the systems are expanded, so is the complexity of the technical solutions. Even if the ASIL of a specific component is the same, the increased complexity means more safety effort, and this typically does not scale linearly. Secondly, expansion of operational situations means more hazardous events with higher severity and exposure. These increases lead to higher ASIL, and thus greater safety effort. Thirdly, the goal is to remove the driver, or at least reduce the driver's involvement in the driving tasks. This removal or reduction causes lower controllability [31], which also means higher ASIL.

The ASIL is rightfully high as an effect of these three mechanisms since it is difficult to refine requirements that are complete with respect to the safety goal, and it is difficult to implement fault free code. High ASIL makes sure that requirement refinement, implementation, verification, and validation are all performed according to strict processes that minimize the risk of faults. However, these processes cause long and costly development. For instance, implementation and verification of high-ASIL software re-

quires, among other things, extensive design reviews of the code and testing of almost every input-output combination to assure that the software requirements are fulfilled to high enough level. But these tasks are cumbersome and require involvement from many people, will still miss faults, and have to be repeated when requirements change [12].

2.3 Saving Some Time

Evidently, the approach of ensuring safety solely by driving many million miles is expensive and time consuming. Hence, there are incentives to limit the amount of miles required to assure a high confidence of correctness. For instance, limiting updates and failures of a system after field tests have started by ensuring that the system is correct from the start saves both time and lives. ISO 26262 provides a processes that can support that aim by judging risks in different situations, and how to mitigate them to acceptable levels. Part of the process is to break down the safety-critical requirements to individual components. The breakdown can be beneficial in the development work since the efforts to ensure safety can start before all components can be integrated into one system and tested as a whole in a field test. However, the extensive reviews and testing needed for components with the highest ASIL are imperfect and expensive in time and money. This is a limiting factor for SAE levels 4 and 5, as there is likely several components with the highest ASIL.

What can be done to construct systems that are correct from the start and limit the efforts put into review and testing? The ISO 26262 standard mentions formal methods as one class of methods that can be used to ascertain correctness of the developed systems. However, the standard does not instruct how formal methods can be applied, so before such methods can be applied within the development process of tactical planners it must be known what problems the formal methods can solve and how to apply them to the applicable problems. This thesis attempts to evaluate how problems in the development of tactical planners in automated cars can be solved by formal methods and how some formal methods compare in terms of benefits and drawbacks. A central question is whether there are any obstacles to adopting formal methods, and especially formal synthesis, as a tool to develop provably correct tactical planners for automated cars.

CHAPTER 3

Formal Methods

Formal methods are a class of methods that make it possible to reason about correctness of systems. A key property of these methods is that requirements and models are expressed in semantics that are very precise and unambiguous.

There are several formal semantic systems. Here we are interested in systems that can express temporal properties since we want to be able to reason about a vehicle's behavior over time. Examples of other systems are systems that reason about the correctness of a program's output only given the current input.

3.1 Automata

Temporal properties can be expressed in formal languages. A formal language is a set of strings constructed from letters of an alphabet according to well-formed and unambiguous rules. Connected to formal languages are finite state automata that can be used to model formal languages.

Finite state automata consists of states and transitions between them. The transitions between the states are labeled with letters from the alphabet and defines which inputs a certain state can consume and how the state changes with that input.

Definition 1: *A finite state automaton is a tuple*

$$A = \langle Q, \Sigma, \delta, q_0, F \rangle,$$

where Q is a finite set of states, Σ is a finite set of symbols, the alphabet, *$q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the accepting states. $\delta : Q \times \Sigma \to Q$ is the transition*

function, a partial function defining which states are connected and which inputs causes these transitions.

In connection with the accepting states F, a finite state automaton has an acceptance condition that states which strings are accepted by the automaton. A string is generated by starting with an empty string in the initial state q_0 and then concatenating $\sigma \in \Sigma$ for every new transition taken. The subset of strings generated this way that fulfills the acceptance condition is the formal language defined by a specific automaton.

3.2 LTL

Linear Temporal Logic (LTL) is a formal logic capable of expressing temporal properties. In addition to atomic propositions in the set AP and standard propositional logic operators, LTL includes temporal operators [32]. The temporal operators $'$ (next), $\Box$ (always), and $\Diamond$ (eventually) are used in this thesis. These temporal operators can be used to express, for instance as done in Paper B, that if one of a vehicle's subsystems fails, then eventually the vehicle shall stop safely at the side of the road.

An LTL formula can be converted into a Büchi automaton that accepts the language expressed by the LTL formula [33], [34]. A Büchi automaton is an automaton that accepts all strings that visit the accepting states an infinite number of times. A Büchi automaton that represents an LTL formula has 2^{AP} as its alphabet — the set of all sets of atomic propositions. Hence, the formal language of an LTL formula is a set of infinite strings, where each letter is a set of the atomic propositions that are true in that particular state.

If LTL formulae are interpreted over infinite runs of a program they can be used to check properties that hold for that program. For this purpose a program is translated into a transition system, of which Kripke structures are one subclass.

Definition 2: *A Kripke Structure is a tuple*

$$M = \langle S, I, R, AP, L \rangle,$$

where S is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function that defines the atomic propositions that are true in each state.

LTL formulae can be evaluated over infinite runs on a Kripke structure. A run π of a Kripke structure M is an infinite sequence of states $\{\pi_0, \pi_1, \dots\}$, where $\pi_0 \in I$ and $(\pi_i, \pi_{i+1}) \in R$. Let $\pi[i]$ represent the infinite run starting from state π_i. Let τ and θ be LTL formulae, and ψ be an atomic proposition. The definition of when a run π satisfies a formula is given inductively:

- $\pi \vDash \tau$ iff $\pi[0] \vDash \tau$
- $\pi[i] \vDash \psi$ iff $\psi \in L(\pi_i)$
- $\pi[i] \vDash \neg\tau$ iff $\pi[i] \nvDash \tau$

- $\pi[i] \models \tau \vee \theta$ iff $\pi[i] \models \tau$ or $\pi[i] \models \theta$
- $\pi[i] \models \tau'$ iff $\pi[i+1] \models \tau$
- $\pi[i] \models \Box\tau$ iff $\pi[k] \models \tau$ for all $k \geq i$
- $\pi[i] \models \Diamond\tau$ iff $\pi[k] \models \tau$ for some $k \geq i$

The Kripke Structure M satisfies a formula τ if every possible run π satisfies the formula.

Two classes of interesting properties of LTL are safety and liveness properties, where the simplest examples are $\Box p$ and $\Box\Diamond p$, respectively. Safety properties can be disproved with finite prefixes of the strings of the formal language, which in principle means that safety properties states what shall never happen. Liveness properties cannot be disproved with finite strings of the language, which means that liveness properties specify conditions that shall be fulfilled infinitely many times.

3.3 Terminology

Foundations for *Model Checking* (MC), *Reactive Synthesis* (RS) and *Supervisory Control Theory* (SCT) are presented in the next sections. The terminologies of Model Checking, Reactive Synthesis, and Supervisory Control Theory are both overlapping and conflicting, as Paper C points out when the formal synthesis tools TuLiP [35] and Supremica [36] are compared, so before they are presented, a note on the terminology used in these fields is needed.

To explain the different parts that go into the different formal methods there are a few terms that need to be defined. What is important for this thesis are mainly three things: *the planner*, *the environment*, and *the requirements*. The planner refers to the (artificial) product that is being designed — usually the tactical planner in the case of this thesis. For a given planning problem, the planner is exactly the parts of a system that are allowed to be changed or designed. The environment is the outside world with which the planner interacts. Within automotive applications this is typically the dynamics of the car, the roads, and other road users. However, the environment also includes other systems and subsystems of the car that cannot or must not be changed during the design of the planner. Examples of this can be brake systems or sensor systems. Lastly, the requirements are the desired behaviors of the system as it interacts with its environment. The requirements define desired and undesired behaviors of the system.

Model Checking starts from a planner, possibly interacting with an environment, and its requirements. However, MC lumps together the planner and its environment and calls them together the *system*. As the name Model Checking implies, MC is the process of checking properties of a model[1] of the planner and, when applicable, the environment. A model is a formal characterization of the planner and environment that captures the behaviors that are interesting for the verification. Depending on the goal of the project the

[1] Also called the system model.

model can be a small piece of code, or an abstraction of an entire car and its surroundings. The formal properties that the Model Checker checks are the formalization of the requirements on the system; the requirements expressed in a formal language [12]. In this thesis the formalization of the requirements will mainly be called the *formal specification*.

In RS, at least in the case of TuLiP, the names have a bit different meaning compared to MC. RS use the concept of environment and system as names for the environment and planner, where the system is constructed such that it guarantees the fulfillment of the requirements as long as the environment model is correct. Also in RS terminology, the system *controls* the environment. Since RS aims to synthesize a controller, or a *reactive module*, there is never a model of the planner in RS. In RS terminology there is neither an environment model, but an environment specification that states the assumed behaviors of the environment. Paper C explains these concepts of RS with some examples. To be consistent through the thesis, the assumed behavior of the environment will be called the environment model for RS.

Supremica, and SCT in general, can be used for both formal verification and synthesis. Regardless, the formalization of the environment, planner, and requirements are divided into plants and specifications. In general, the plants are the models of the environment and/or planner, whichever is applicable. The specifications are the formalizations of the requirements. However, the plants can also have markings on states, which is a formalization of a certain type of requirements that express the possibility to reach those states. Some examples of this are also given in Paper C.

3.4 Model Checking

Model Checking is used to prove formal properties of a formal model of the system. The system can consist of hardware or software, but it is software applications that are considered here. Verification of software with MC starts with the formal specification, which is composed of the formal properties, and the formal model. The intended result of model checking is either a proof that the model exhibits the formal property, or a counterexample that demonstrates why the property does not hold in the model. The benefit of performing model checking over reviewing and testing is that model checking is exhaustive and can prove that no faults are present. However, a caveat is that this holds only when the model captures the relevant behaviors of the system. The model checker (the program that performs the Model Checking) can also run out of time or memory because of the state explosion problem. Then model checking can only be used to find faults, just like testing.

In LTL Model Checking the formal properties are expressed in LTL and the software program is modeled as a transition system.

Definition 3: *Let* $K = \langle S, I, R, AP, L \rangle$ *be a transition system and* $B = \langle Q, Q_i, \delta, AP, F \rangle$

be a Büchi automaton. Then the product $K \times B$ is the following tuple

$$T = \langle S \times Q, N_i, \rightarrow, AP, N_a \rangle,$$

where

$$\begin{aligned}(s,q) \rightarrow (s',q') \quad & \textit{iff } (s,s') \in R \textit{ and } (q, L(s'), q') \in \delta \\ N_i = \{(s_i, q) \,|\, s_i \in I \ & \textit{and } \exists q_i \in Q_i : (q_i, L(s_i), q) \in \delta\} \\ & N_a = \{(s,q) \,|\, q \in F\}\end{aligned}$$

Let the transition system K be a Kripke structure that models a software program, and φ an LTL formula that expresses a desired formal property of K. To prove whether K satisfies φ the following steps are performed [12]:

(i) Construct a Büchi automaton B for the LTL formula $\neg\varphi$.

(ii) Construct the product $K \times B$.

(iii) Analyze whether $K \times B$ has a run π visiting accepting states infinitely often.

(iv) If such a π is found, then φ is not satisfied by K, and π is a counterexample. If no such π is found, then K satisfies φ.

The construction of transition systems and Büchi automata can be quite involved, so there are software tools that let the user define the transition system in a formal format that is more suitable for humans, and the properties in LTL. These are then automatically translated into a transition system and Büchi automata, and then turned into a product tuple, which in turn is searched for the existence of π. One such model checker is Spin [37], which allows the user to specify the system model in the language Promela, and the desired properties in LTL.

3.5 Reactive Synthesis

Reactive Synthesis (RS) [14] aims to automatically synthesize a reactive module that satisfies the desired *guarantees* ϕ_s, under the *assumptions* of the environment ϕ_e. In other words, the reactive module satisfies the formula $\phi_e \rightarrow \phi_s$ [38]. A reactive module evolves in computation cycles and engages in an ongoing interaction with its environment. The reactive module alternately reads inputs from the environment and assigns values to its outputs, possibly affecting the environment. For a comprehensive introduction to RS, refer to [41].

One way to model the RS problem is to consider the reactive module to be synthesized and the environment as adversaries that play a finite-state game and take turns to provide input to each other [42]. Then, an iterative process can be adopted to find a fix-point

of a subset of states and transitions that solves the RS problem. The states, transitions, inputs, and outputs can be modeled by a Kripke structure.

In RS, the set of atomic propositions AP is divided into two disjoint subsets AP_e and AP_s, representing the propositions of the environment and the reactive module, respectively. The atomic propositions in AP_e are seen as the inputs to the reactive module, while AP_s are its outputs. The atomic propositions in AP can be composed of relational operators on functions of discrete finite domain variables so they can be considered either true or false, given a certain valuation in a certain state.

A subset of LTL formulae that is particularly interesting is called *General Reactivity of Rank 1* (GR(1)) [43]. This set of formulae is useful because it is possible to perform reactive synthesis of the GR(1) fragment in polynomial-time. Given an LTL formulae φ defining the environment model and the formal specification, GR(1) synthesis constructs a reactive module such that the total system satisfies φ, if such a reactive module exists. The LTL fragment GR(1) has the following form [43]:

$$\varphi \triangleq \big((\psi^e_{init} \wedge \Box\psi^e_{safe} \wedge \bigwedge_{0<i\leq J}\Box\Diamond\psi^e_{live,i}) \rightarrow (\psi^s_{init} \wedge \Box\psi^s_{safe} \wedge \bigwedge_{0<i\leq N}\Box\Diamond\psi^s_{live,i})\big), \tag{3.1}$$

where ψ^e_{init} and ψ^s_{init} contain all the initial conditions of the environment and the reactive module, respectively. ψ^e_{safe} and $\psi^e_{live,i}$ (J number of sub-specifications) are the *safety* and *liveness* assumptions on the environment. ψ^s_{safe} and $\psi^s_{live,i}$ (N number of sub-specifications) are the desired safety and liveness properties of the reactive module.

Note that the implication in (3.1) shall be interpreted as a *strict realizability* implication [44]. Strict realizability means that the reactive module must not 'cheat' by falsifying the environment assumptions. If a normal logical implication would be used, the reactive module could be synthesized such that it violates the formal specification to force the environment to violate the assumptions. If any of the formulae on the left of the implication in (3.1) is false, then the implication is trivially true if it is not interpreted as a strict realizability implication.

Klein and Pnueli give the following counterexample with two boolean variables p and q, where p is an environment variable and q is a reactive module variable:

$$\Box\neg p' \wedge \Box\Diamond(p \leftrightarrow q) \rightarrow \Box(p' \leftrightarrow q') \wedge \Box\Diamond q. \tag{3.2}$$

If the implication in (3.2) is treated as the standard logical implication, a reactive module could satisfy the formula by letting q be True in all states. The environment then cannot fulfill both its assumptions, and thus the implication is satisfied regardless of whether the reactive module fulfills its specifications or not.

The strict realizability implication details that the reactive module must not be the first player to violate the safety properties. Formula (3.2) cannot be satisfied if the implication is strictly realizable. If the environment fulfills the safety assumptions then p is False all the time, and hence the reactive module must let q be False all the time, otherwise breaking the specification $\Box(p' \leftrightarrow q')$. The liveness assumption $\Box\Diamond(p \leftrightarrow q)$ is satisfied, but the reactive module cannot then fulfill the liveness specification $\Box\Diamond q$.

3.6 Supervisory Control Theory

The Supervisory Control Theory (SCT) [13], [45] is a model-based approach for control of Discrete Event System (DES). A DES is a category of automata whose transitions are triggered by occurrences of *events*; the labels on the transitions are not interpreted as inputs but observations of events. Given a DES to be controlled, the *plant*, and a *specification* describing the desired behavior, a control entity, called a *supervisor*, can be automatically synthesized to dynamically restrict the behavior of the plant, such that the closed-loop system satisfies the specification.

Finite state machines (FSM) are defined the same as automata (1), but is given here again to stress that the alphabet consists of events.

Definition 4: *An FSM is a tuple*

$$A = \langle \Sigma, Q, \rightarrow, Q^{\circ}, Q^{\omega} \rangle,$$

where Σ *is a set of events, the* alphabet*;* Q *is a finite set of* states*;* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the* transition relation*;* $Q^{\circ} \subseteq Q$ *is the set of* initial states*; and* $Q^{\omega} \subseteq Q$ *is the set of* marked states*.*

In SCT, the accepting states are termed the marked states, so Q^{ω} are equal to F in (1). The accepting condition of an FSM accepts all *finite* strings that are a prefix to a string ending in a marked state [13][2].

Given a plant G and a specification K of the desired controlled behavior, the SCT makes it possible to automatically synthesize a supervisor S that controls the plant such that the specification is satisfied.

The supervisor and the plant form an asymmetric *closed-loop* system where the supervisor restricts the event generation of the plant. As the plant generates events enabled by the supervisor, the supervisor observes the generated sequences of events and determines, at each state, which of the currently possible events that should be enabled. Some of the events cannot be disabled by the supervisor. These events are *uncontrollable.* The supervisor is required to be *controllable*, i.e. it must never try to disable an uncontrollable event. By disabling controllable events, the supervisor can confine the plant to a subset of its possible states so that the closed-loop system only visits states that are considered “good” by the specification. At the same time, the supervisor guarantees that from any closed-loop state, the system can always continue to a desired marked state, it is *non-blocking.* Finally, the supervisor disables as few events as possible, it is *maximally permissive.*

For a plant with controllable and uncontrollable events, the alphabet Σ consists of the two disjoint sets Σ_c and Σ_u consisting of the controllable and uncontrollable events, respectively.

Definition 5: *Given two FSM* $G_1 = \langle \Sigma_1, Q_1, \rightarrow_1, Q_1^{\circ}, Q_1^{\omega} \rangle$ *and* $G_2 = \langle \Sigma_2, Q_2, \rightarrow_2, Q_2^{\circ}, Q_2^{\omega} \rangle$*, the* full synchronous composition *of* G_1 *and* G_2 *is* $G_1 \parallel G_2 = \langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \rightarrow,$

[2]There are also SCT variants with accepting conditions that only accepts infinite strings [46]. Then the accepting condition is the same as for Büchi automata.

$Q_1^\circ \times Q_2^\circ, Q_1^\omega \times Q_2^\omega\rangle$, *where:*

$$\begin{aligned}
(x_1, x_2) \xrightarrow{\sigma} (y_1, y_2) \quad & \textit{if } \sigma \in \Sigma_1 \cap \Sigma_2,\ x_1 \xrightarrow{\sigma}_1 y_1, \\
& \textit{and } x_2 \xrightarrow{\sigma}_2 y_2 \ ; \\
(x_1, x_2) \xrightarrow{\sigma} (y_1, x_2) \quad & \textit{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \textit{ and } x_1 \xrightarrow{\sigma}_1 y_1 \ ; \\
(x_1, x_2) \xrightarrow{\sigma} (x_1, y_2) \quad & \textit{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \textit{ and } x_2 \xrightarrow{\sigma}_2 y_2 \ .
\end{aligned}$$

Basically, supervisor synthesis is an iterative removal of states and/or transitions of an initially calculated supervisor “candidate”. Practically, this candidate is calculated from $G||K$ where $||$ denotes the *full synchronous composition* operator [13]. The iterative algorithm removes from $G||K$ the states that break the controllability and/or the non-blocking properties. Iteration is necessary since enforcing one property may break the other. The iteration will eventually reach a fix-point, and what then is obtained is the maximally permissive, controllable, and non-blocking supervisor.

Extended finite-state machines (EFSM) are similar to FSM, but augmented with *updates* associated to the transitions [47], [48]; formulas constructed from variables, integer constants, the Boolean literals *true* (**T**) and *false* (**F**), propositional logic connectives, and discrete arithmetic operators.

A *variable* v is an entity associated with a bounded discrete domain $\mathrm{dom}(v)$ and an initial value $v^\circ \in \mathrm{dom}(v)$. Let $V = \{v_0, \dots, v_n\}$ be the set of variables with domain $\mathrm{dom}(V) = \mathrm{dom}(v_0) \times \cdots \times \mathrm{dom}(v_n)$. An element of $\mathrm{dom}(V)$ is called a *valuation* and is denoted by $\hat{v} = (\hat{v}_0, \dots, \hat{v}_n)$ with $\hat{v}_i \in \mathrm{dom}(v_i)$, and the value associated to variable $v_i \in V$ is denoted $\hat{v}[v_i] = \hat{v}_i$. The *initial valuation* is $v^\circ = (v_0^\circ, \dots, v_n^\circ)$.

A second set of variables, called *next-state variables*, denoted by $V' = \{\, v' \mid v \in V \,\}$ with $\mathrm{dom}(V') = \mathrm{dom}(V)$, is used to describe the values of the variables after execution of a transition. Variables in V are referred to as *current-state variables* to differentiate them from the next-state variables in V'. The set of all update formulas using variables in V and V' is denoted by Π_V.

For an update $p \in \Pi_V$, the terms $\mathrm{vars}(p)$ and $\mathrm{vars}'(p)$ denote the sets of all variables, and all next-state variables, respectively, that occur in p. For example, if $p \equiv x' = y + 1$ then $\mathrm{vars}(p) = \{x, y\}$ and $\mathrm{vars}'(p) = \{x\}$. Here and in the following, the relation $\equiv$ denotes syntactic identity of updates to avoid ambiguity when an update contains the equality symbol $=$.

With slight abuse of notation, updates $p \in \Pi_V$ are also interpreted as predicates over their variables, and they evaluate to **T** or **F**, i.e., $p\colon \mathrm{dom}(V) \times \mathrm{dom}(V') \to \{\mathbf{T}, \mathbf{F}\}$. For example, if $V = \{x\}$ with $\mathrm{dom}(x) = \{0, 1\}$, then the update $p \equiv x' = x + 1$ means that the value of the variable x in the next state will be increased by 1 over its current-state value. Its predicate $p(x, x')$ evaluates as $p(0, 1) = \mathbf{T}$ and $p(1, 1) = p(0, 0) = p(1, 0) = \mathbf{F}$.

Definition 6: *An* extended finite-state machine (EFSM) *is a tuple*

$$E = \langle \Sigma, Q, \to, Q^\circ, Q^\omega \rangle,$$

where Σ *is a set of events, the* alphabet*;* Q *is a finite set of* locations*;* $\rightarrow \subseteq Q\times\Sigma\times\Pi_V\times Q$ *is the* conditional transition relation*;* $Q^\circ \subseteq Q$ *is the set of* initial locations*; and* $Q^\omega \subseteq Q$ *is the set of* marked locations*.*

The expression $q_0 \overset{\sigma:p}{\rightarrow} q_1$ denotes the presence of a transition in E, from location q_0 to location q_1 with event σ and update p. Such a transition can occur if the EFSM is in location q_0 and the update p evaluates to $\mathbf{T}$, and when the transition occurs, the EFSM changes its location from q_0 to q_1 while updating the variables in $\mathrm{vars}'(p)$ in accordance with p; variables not contained in $\mathrm{vars}'(p)$ are unchanged.

Given an EFSM $E = \langle\Sigma, Q, \rightarrow, Q^\circ, Q^\omega\rangle$, its *variable set* is

$$\mathrm{vars}(E) = \bigcup_{(q_0,\sigma,p,q_1)\in\rightarrow} \mathrm{vars}(p),$$

which contains all variables that appear on some transitions of E.

Definition 7: *Given two EFSMs* $E_1 = \langle\Sigma_1, Q_1, \rightarrow_1, Q_1^\circ, Q_1^\omega\rangle$ *and* $E_2 = \langle\Sigma_2, Q_2, \rightarrow_2, Q_2^\circ, Q_2^\omega\rangle$*, the* synchronous composition *of* E_1 *and* E_2 *is* $E_1 \parallel E_2 = \langle\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \rightarrow, Q_1^\circ \times Q_2^\circ, Q_1^\omega \times Q_2^\omega\rangle$*, where:*

$$\begin{aligned}
(x_1, x_2) &\xrightarrow{\sigma:p_1\wedge p_2} (y_1, y_2) \quad \text{if } \sigma \in \Sigma_1 \cap \Sigma_2,\ x_1 \xrightarrow{\sigma:p_1}_1 y_1, \\
&\qquad\qquad\qquad\qquad \text{and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ ; \\
(x_1, x_2) &\xrightarrow{\sigma:p_1} (y_1, x_2) \quad \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } x_1 \xrightarrow{\sigma:p_1}_1 y_1\ ; \\
(x_1, x_2) &\xrightarrow{\sigma:p_2} (x_1, y_2) \quad \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ .
\end{aligned}$$

In an EFSM, the current state of the system is given by the current location together with the current values of all the variables. Since the variables are discrete and bounded, the EFSM can be *flattened*, which introduces states for the combinations of locations and variable values [12].

Definition 8: *Let* $E = \langle\Sigma, Q, \rightarrow, Q^\circ, Q^\omega\rangle$ *be an EFSM with variable set* $\mathrm{vars}(E) = V$*. The* monolithic flattening *of* E *is* $U(E) = \langle\Sigma, Q_U, \rightarrow_U, Q_U^\circ, Q_U^\omega\rangle$ *where*

- $Q_U = Q \times \mathrm{dom}(V)$*;*
- $(x, \hat{v}) \overset{\sigma}{\rightarrow}_U (y, \hat{w})$ *if* E *contains a transition* $x \overset{\sigma:p}{\rightarrow} y$ *such that* $p(\hat{v}, \hat{w}) = \mathbf{T}$*;*
- $Q_U^\circ = Q^\circ \times \{v^\circ\}$*;*
- $Q_U^\omega = Q^\omega \times \mathrm{dom}(V)$*.*

$U(E)$ is the DES representation of the EFSM, where all the variables have been removed and their values $\hat{v}$ embedded into the state set Q_U. This ensures the correct sequencing of transitions in the DES.

With the introduction of updates to FSMs it is possible to restrict the plants' behaviors by adding updates to the plants. In that case the synthesis algorithm does not need to generate a supervisor as separate automata.

For the purpose of adding updates to plants, *Binary Decision Diagram* (BDDs) [49] can be used in the synthesis algorithm. A BDD is a data structure that can efficiently store huge state-spaces and transition sets encoded as Boolean functions. The computational complexity of synthesis using BDDs does not depend on the number of states or transitions, but on the number of nodes in the BDD, as the computations are performed *symbolically* rather than explicitly; synthesis is performed on sets of states and transitions rather than single such elements. So even though the BDD-based synthesis works on a monolithic model, it can handle systems of considerable sizes. The approach presented in [50] uses partitioning techniques to further stretch the ability of the synthesis procedure.

The BDD-based synthesis works directly on the EFSM model, encoding the transition relation and the updates as Boolean expressions. The synthesis then results in a partitioning of the state-space into *allowed* and *forbidden* states, and then guarantees that only controllable events take the system from the allowed into the forbidden states. From this is then generated updates that represent the control actions of the supervisor. These updates are Boolean expressions over values of variables and state labels, that define when the particular event is enabled.

CHAPTER 4

Correct-by-Construction Tactical Planners

In theory, formal methods are a promising approach for the development of correct tactical planners, and case studies have been performed to demonstrate the real usefulness of formal methods applied to practical problems [51]–[53]. Tactical planners are special because of three characteristics;

- they take discrete decisions in environments where they interact with other road users;
- the car's position continuously changes and therefore they are moving while taking decisions; and
- they need to be agnostic of absolute position since they should operate in large parts of the world's road network.

These three characteristics mean special consideration because temporal specifications are needed to describe the desired interactions, and the states need to have a careful abstraction.

This chapter describes how the characteristics affect the modeling of the planner and the environment, and the formalization of the requirements. Especially, different modeling formalisms are compared to expose benefits and drawbacks of using one formalism over the another.

4.1 Inspection of Results

One of the key findings of obstacles to using formal synthesis in larger scale industrial applications is the importance of inspection of the results. When the number of states of the supervisor or reactive module exceeds about 20 and when the states are highly interconnected, it becomes very difficult to inspect and understand their resulting properties. Obviously, the supervisor or reactive module adheres to the stated requirements that all have very precise and formal syntax and semantics, but with several interacting requirements it is increasingly difficult to anticipate all effects of the interactions. An objection would be that the supervisor or reactive module could be calculated by hand to get an understanding of the results, but that defeats the purpose of applying automatic methods to reduce the efforts of manual labor.

Interestingly, this obstacle of interpreting the results are also an issue within machine learning (ML), and specifically within deep learning (DL) [54], [55]. On a high level the work processes of ML and formal synthesis are very similar. A model of the environment is one input (data in the case of ML and a formal model in the case of formal synthesis), the requirements is the second input (reward function for ML and formal specification for formal synthesis), and a black box model is the output (a trained agent in ML and an automaton in formal synthesis). The goal in both fields is to find a black box output that fulfills the requirements, adhering to the rules of the environment.

Although the solutions to these issues of explainability or ability to inspect and understand most likely require different tools for analysis and interpretation, the interest of the topic in ML still indicates how important these issues are to solve before the methods can be used with credibility; explaining why and when they work are important when arguing that automated cars are safe in certain operational domains. As will be explained in Section 4.2 though, formal synthesis has benefits in that the assumptions on the environment become very clear because of the separation of environment and specification.

The problem of inspection is a large part of Paper A where the results would otherwise not have been possible to understand. Paper D also includes examples where the possibility to inspect the black box result is important in order to formalize the correct environment model and the correct requirements. In Paper C the focus was to compare the synthesis methods from a modeling perspective, and thus the importance of inspection was not made explicit. However, conclusions of the issue of inspection can still be drawn from the results of Paper C. Formal verification does not produce a black box, and hence Paper B does not cover the issue, but the results from that paper can still be used to reason about the usefulness of methods whose model and requirements are easily interpretable.

The goal of Paper A is to patch a fault in a manually implemented tactical planner that is responsible for deciding when it is safe to make lane changes (Lateral State Manager (LSM), shown in Figure 1 in Paper A). The manually implemented code was modeled by

Zita *et al.* [16] and formally verified with Supremica [36]. The result of the verification exposed several faults, of which one was very elusive in the attempts to manually correct it. Paper A attempts to automatically produce a patch for the LSM by using Supremica to formally synthesize a supervisor.

The supervisor obtained for the LSM this way is correct by construction, but it is clear that 'correctness' in this case does not mean the same as patching and correcting the LSM. To come to this conclusion — that the supervisor did not rectify the fault within the LSM — requires inspection of the supervisor.

Supremica can generate supervisors as automata in several different ways. The simplest automaton to analyze in the scope of Paper A consists of 700 states, and is unreasonable to visually inspect. Another option for interpreting it can be to simulate it, but that defeats the purpose of automatic synthesis.

However, Supremica can also use BDD-based synthesis, where the result is added as updates to the existing plants instead of generating new automata[1]. It is shown in Paper A that the BDD-based supervisor is equivalent to the automata-based supervisors. It is also evident that the updates added by the synthesis do not have the intended effect of patching the fault; instead of correcting the fault the guards put limitations on which values the inputs to the LSM can take. This issue can be traced back to the model, but would not have been found easily if it were not for the possibility to inspect the generated supervisor.

Paper D further indicates the importance of interpreting the synthesis results. By inspection, errors are found in both the model of the environment and in the formalizations of the requirements. In contrast to Paper A, that only concerns SCT (more specifically Supremica), Paper D also brings these conclusions into the field of RS, or at least synthesis with the GR(1) fragment, by evaluating synthesis in TuLiP [35] side-by-side with Supremica [36]. Given that booth tools implement fundamental algorithms in their field, it seems likely that the results on inspection applies broadly in the fields.

In addition to finding errors in the formalizations, Paper D finds that inspection of the supervisor and the reactive module is an important tool for qualitatively comparing the properties of them. For instance, it is possible to visually inspect and conclude that the supervisor allows all the traces that the reactive module accepts, but the reactive module does not accept all traces that the supervisor allows.

In its first case study, Paper C similarly use visual inspection of the supervisor and the reactive module to ascertain that they represent the same solution. They both have relatively small state spaces, so visual inspection is effective.

The supervisor and the reactive module for the second case study in Paper C, on the other hand, have huge state spaces that are unsuitable for visual inspection, even if the length of the road is reduced heavily. The comparisons in Paper C that require evaluation of the results are instead performed with simulations. Simulations are enough to obtain

[1]The BDD-based synthesis is convenient when patching since the updates are easy to translate to if-statements and add to the original code.

the results and draw the conclusions presented in Paper C, but better tools for inspection might have contributed to more profound results.

As already stated above, Paper B does not perform synthesis. However, code and abstract state machines are used in the design to aid the understanding of the behaviors of the tactical planner. Compared to the generated supervisor and reactive module, the code and abstract state machines seem to facilitate understanding by providing structure and by merging (or abstracting) states that have the same behavior.

The same idea might be applied to the synthesized planners' states to simplify the inspection of large supervisors and reactive modules. State transitions of planners that have no effect on the current output might contribute to an unstructured representation in the synthesized planners' automata-representations, and merging those states might benefit the understanding.

4.2 Verification vs. Synthesis

Formal verification is a versatile tool for proving absence of faults. This is experienced in Paper B, which uses Model Checking simultaneously with model-based design in an iterative process to develop a tactical planner for an automated car. As discussed in Section 3.3, the model of the planner and the environment are merged and there is no precise distinction between the two. The formal specification of desired properties is entirely in LTL. As noted in Paper B, the implementation of the tactical planner was easy to integrate with the rest of the car's systems, and no errors were found during simulation or in-car experiments. Although the simulations and in-car experiments were not exhaustive, the smooth integration gives some indication that the process with alternating design, implementation, and model checking contributes to less errors.

However, formal verification has some drawbacks: Paper B concludes that the model of the planner has to be updated to match new implementations of the planner. Manual modeling always has the possibility of introducing faults [12], and remodeling naturally increases that risk. The modeling can in some instances be made automatic, but then the same caveat applies for the translation algorithm. Another drawback, experienced by Zita *et al.* [16], is when a specification does not hold and the formal verification reports a counterexample that is difficult to remedy manually.

Paper A and Paper D are similar to each other and investigate how the formal models and requirements already derived by Zita *et al.* and Paper B, respectively, can be used as input to a synthesis algorithm. The main conclusion of both Paper A and Paper D is that it is helpful to already have formal models and requirements, but they need to be reworked before they can be successfully used in synthesis.

Zita *et al.* [16] provide plants and specifications in Supremica to find a fault in the LSM. The approach of Paper A is to use the same plants and specifications as input to formal synthesis to generate a patch for the LSM. This approach fails, and the main reason seems to be that the verification model does not have any separation of the planner and

the environment. All dynamics are treated as a description of the environment behavior, where the planner can restrict *all* behaviors. The verification model is a direct translation of the code, so every line of code is represented, but for synthesis what matters is how the inputs behave, how that affects the state, and how the outputs should be set based on the inputs and the state.

Paper D illustrates more distinctly what type of changes are needed to go from verification to synthesis. As noted above, the verification model has merged the environment and planner behaviors, and they have to be split before the synthesis can start. To start from such a pre-existing model of the environment simplifies the modeling for synthesis, but there is a significant difference in syntax and semantics in the imperative verification model in Promela on one hand, and the declarative LTL and automata in TuLiP and Supremica on the other.

From an operational domain perspective the distinct separation of environment behaviors and planner behaviors is a great benefit of the synthesis process. As mentioned above in Section 4.1, explaining when the synthesized planner works is an important part of a credible safety argument [10], and separate formal environment assumptions can simplify that argumentation.

The specification expressed in LTL in Paper B can be used with almost no changes in the synthesis of the reactive module in Paper D. Minor changes are needed to account for the limitations that GR(1) brings, for instance $\Diamond\Box$ is not supported in the formal specification directly. The translation of the LTL formulae into automata for Supremica is more involved, but is not an obstacle. However, the LTL specification of Paper B is not complete; it only expresses the properties that are most interesting to verify. Hence, more requirements are needed to synthesize a tactical planner with the same behavior as the manually implemented one.

The model of the planner used for verification is not used directly in synthesis because it is the part to be generated, but if available it guides the formalization of the requirements on the planner. If the tactical planner of Paper A would be synthesized, then it too will probably suffer from incomplete requirements. That synthesis with one formal specification could lead to the desired tactical planner is unlikely. A successful synthesis of the tactical planner as described in Paper A requires a separation of the planner and environment model as laid out in Paper D, and the planner model should be used to guide the formalization of a complete set of formal specifications.

The incompleteness of the requirements in formal verification is only a drawback when they are later used in synthesis, however; when the only purpose is to formally verify the planner and environment, the possibility to verify only the properties that matters simplifies the process.

4.3 Abstraction of States

Cars move in continuous time and space, but tactical planners make discrete decisions at discrete time instances. The continuous time and space of the physical car must be discretized or abstracted into discrete states before the formal methods described here can be used to prove properties of the planners or to synthesize them. The abstraction can be done to different levels. For instance, Paper B and Paper C abstract a car's longitudinal dynamics[2] in fine detail (or low abstraction), where distance is divided into many small steps. The discrete speed determines how many steps the car moves in every time step, just like the dice decide how many streets the players move in Monopoly[3].

A major advantage of a low level of abstraction is the type of formal specifications that can be expressed. If the model has many details it is also possible to express detailed specifications. Both Paper B and Paper C display this advantage. In Paper B it is possible to express a specification for 'do not pass the end of a path', and in Paper C it is possible to express that the two automated cars shall 'keep a safety distance'. Both of these specifications are intuitive to state as they refer to end effects.

In the end, what is important for the safety of tactical planners is that they are safe, which often means that they must not crash. The low level of abstraction used in Paper B and Paper C means that such 'do not crash'-requirements can be expressed directly.

The detailed models of Paper B and Paper C come with a price, however. Their low level of abstractions are infeasible from two perspectives.

Firstly, the correctness of the planners is restricted to the certain model of the environment, which in Paper B and Paper C means the specific lengths of the paths and the specific length of the circular road with one other car, respectively. In practice, a car is used in much more varied situations than what is modeled in Paper B and Paper C. Even if an automated car has a very restricted operational domain, there are far too many roads and traffic occupants to model them all. In conclusion, the tactical planners must be generic and fit a wide range of environments with no dependence on detailed states. Since Paper B is using model checking in the development of the planner, it is possible to design a generic planner and verify it on a detailed model. The planner can handle several different paths and is verified on one detailed path. An induction argument could be made for its correctness on other paths. Paper C, however, uses synthesis to generate a planner, and that planner's decisions depend on both cars' positions on the road. Removing the other car or increasing the number of steps of the road after synthesis has been performed leads to invalid states, and the decisions of the planner become nonsense.

Secondly, the low level of abstraction means a large number of states. Even if the operational domain is limited, there will be a huge number of states. Formal methods suffer from the state explosion problem, which means that the memory needed to store all states becomes too big to be tractable. For any reasonably capable tactical planner

[2] That is, the dynamics in the driving direction. Paper B considers the motion along paths and Paper C considers a straight road.

[3] And just like in Monopoly, something bad happens when moving too fast.

the state explosion problem would be a serious obstacle with a low level of abstraction. The paths in Paper B has a maximal length of 1500, and Spin cannot handle much more. Supremica is close to the memory limit when the road in Paper C is 100 steps long.

Paper A and Paper D aim to synthesize generic planners that are not bound to specific positions as in Paper B and Paper C. To accomplish that, their environment models have a higher level of abstraction. Neither of the two papers have states that represent detailed position, but rather the states represent operational modes. Paper D, for instance, does not include position along a path or the car's speed in the model of the environment. Instead the model considers all positions along one path as one state, and the car can either be stopped or driving. This higher level of abstraction leads to generic planners since the details of the paths in Paper D does not matter to the planner, as long as the process of passing between them is the same.

In addition to the benefit of allowing generic planners, synthesis with high levels of abstraction also decreases the required state space. Then larger problems can be synthesized, and the synthesized planners become easier to inspect and interpret if they consist of fewer states.

In summary, the level of abstraction is a very important consideration when applying formal methods. The level of abstraction will determine what type of requirements can be expressed, how large problems can be handled, and how generic the tactical planner can be. Ideally, we would like to express detailed requirements, solve large problems, and design generic planners. What should the models look like to accomplish such goals?

4.4 Modeling Considerations

As seen before, the level of abstraction of the models is important to consider before applying formal methods to a design problem. The abstraction is an important part of the model in any language, but there are also important considerations as to which modeling formalism to model in. As already indicated in Chapter 3, the different formal languages have many similarities in their underlying definitions and algorithms, but their semantics and what is possible to express can differ widely. However, despite these differences, it can often be possible to acquire similar results in the different formalisms by careful modeling. This section will mainly cover the differences in modeling for synthesis.

Paper C and Paper D study differences in modeling in RS and SCT, specifically differences between TuLiP and Supremica. One fundamental difference between TuLiP and Supremica is the modeling format. TuLiP has support for LTL synthesis, albeit a restricted fragment of LTL, and Supremica's modeling is performed with automata. Although writing LTL or drawing automata is a major difference in terms of modeling language and for the process of the designer, it does not seem to matter in terms of what problems and solutions that formal synthesis can be applied to, at least not in the problems investigated in the appended papers. Based on the theory of formal languages in Chapter 3 and the results in Paper C and Paper D, it seems like the modeling formats

are sufficiently similar to allow expressions of similar structure in an automaton sense.

The GR(1) fragment, which the TuLiP model is restricted to, limits which formulae that can be used in the model. Although GR(1) excludes a large part of LTL, it is still possible to specify such properties with a combination of GR(1) formulae and extra atomic propositions in AP. Piterman *et al.* [43] demonstrate how 'until' can be specified in GR(1) synthesis, and Paper D has examples of how other non-GR(1) formulae are expressed. For instance, the property of $\Box(p \rightarrow \Box q)$, where $p, q \in AP$, can be expressed as:

$$\Box(q \rightarrow q') \tag{4.1}$$

$$\Box(p \rightarrow q) \tag{4.2}$$

The property $\Diamond\Box p$ is also not part of GR(1), but is easy to express in GR(1) synthesis by the formulae

$$\Box p \rightarrow p \tag{4.3}$$

$$\Diamond p \tag{4.4}$$

Hence, as indicated by both Paper C and Paper D, the restrictions of GR(1) does not seem to be an obstacle for synthesizing tactical planners in the considered problems. Whether these results are general is not clear and needs to be evaluated by a more systematic review of the type of requirements and environments that applies to a broader class of tactical planners.

The GR(1) restriction of LTL does not seem to be an obstacle for the modeling compared to LTL, but the difference between the properties that can be expressed in LTL in general and in SCT can become obstacles for synthesis. Hence, it is important to select a suitable modeling formalism depending on what type of problem that should be solved and what kind of solution that is desired.

In Section 3.6 it is stated that supervisors synthesized with SCT are maximally permissive; the supervisor disables as few events as possible, while still guaranteeing that all requirements are fulfilled. This characteristic of supervisors is unparalleled in RS, so its effects are important for the choice of synthesis formalism. Maximal permissiveness allows supervisors to restrict unsafe behaviors, while the supervisors do not mandate which safe behaviors that should be promoted. Depending on problem and desired solution a maximal permissive planner can be a benefit or a drawback. The intention of having maximally permissive supervisors is that they shall restrict a system to the safe states and not interfere with decisions that are not affecting safety. However, that intention does not mean that synthesized supervisors are not allowed to be applied to other problems, as illustrated in [56].

Starting with the simple game modeled in Paper C, it is shown that maximal permissiveness does not always mean a difference when the supervisor is compared to a reactive module. If there is only one solution to a problem the two formalisms generate the same

planner. However, it needs to be considered that the number of possible solutions might not be known a priori.

In Paper A, the maximal permissiveness property of the supervisor were to be used as a restriction on the already implemented software. Instead of implementing a new planner based on a synthesized supervisor it might have been used to generate additional restrictions on the software to ensure its safety. Since no supervisor was synthesized as intended we can only speculate on the plausibility of such restrictions.

The tactical planners synthesized by Supremica in Paper C and Paper D show the effects that maximal permissiveness has. In many states of these supervisors several controllable events are allowed to fire. It is up to an *event generator* to fire one of them, or in some cases wait for an uncontrollable event to fire. The supervisors give no indication of whether one event is better than another, only that they all keep the system in a safe set of states if fired. The need for an event generator can be seen as an opportunity or a drawback. If the supervisor is intended to be a restriction that maintains a vehicle in a safe set of states, then it might be beneficial to only formalize the safety related requirements. If it is difficult to formalize which choice is best among the safe actions, an event generator can then be developed that selects among the safe choices. The event generator is, on the other hand, another decision maker to develop. There are extensions of SCT that can automatically create an event generator [56], and modeling techniques can be used to limit the number of enabled events [57]. These methods affect the modeling, so whether an event generator is desired that should be known a priori.

Reactive modules are not maximally permissive. The reactive modules that are synthesized by TuLiP in Paper C and Paper D have one certain output for each input. Hence, the reactive modules do not need an event generator and can directly be implemented as tactical planners. This means that they have a defined behavior and do not represent only a restriction of the actions, they define exactly what decision to make. A caveat of this characteristic is, as noted in Paper D, that behaviors that are not specified can be realized. So, consider a formal model and specification that are synthesized into a tactical planner that exhibits a desired behavior even if that behavior is not specified. Consider also that the tactical planner later must be augmented with a new requirement not relating to the original desired behavior. Since the original desired behavior is not specified there is a chance that it disappears in synthesis with the new requirement.

As Paper D illustrates, a maximally permissive supervisor *will* display such specification omissions while a reactive module *might* show them. As discussed in Section 4.1, it is difficult to understand what properties all the interactions between the specifications result in. Maximally permissiveness gives a possibility to view what properties that are specified or not.

Further considerations when choosing between synthesis formalisms are progress, cycles, and turns. Progress means that the planner and the environment are 'moving' toward the marked states. In TuLiP, progress is synonymous with liveness, but the term progress is used in this context since liveness has a strict definition and is not defined for

Supremica.

The accepted languages of TuLiP and Supremica differ in two important aspects related to progress: the length of the strings, and whether marked states must appear in them. TuLiP builds upon the semantics of LTL, which means that the language of the environment and the formal specification consists of infinite strings of states, where marked states appear infinitely often (see Section 3.2). With $\Box\Diamond$ it is therefore possible to specify that a system *shall* reach a certain state, and the synthesized reactive module *guarantees* that the state is reached. Supremica, on the other hand, are based on FSM that only accept finite strings of states. These accepted strings of states either end with a marked state, or are prefixes of such strings; the language contains finite strings that does not include the marked states (see Section 3.6). What this means for the modeling semantics in SCT is that a system *may* reach a marked state, and the synthesized supervisor must *allow* at least one marked state to be reached.

The specifications of Supremica cannot express progress, but the the maximal permissiveness of a supervisor means that the structure of its FSM representation allows all safe progress properties. In a sense, a supervisor allows all safe progress properties by default, while a reactive module generated by TuLiP is only guaranteed to satisfy the specified progress properties.

Paper C discuss cycles in connection to progress. Cycles are sequences of states that are repeated forever. They are easy to express in TuLiP, as shown in Paper C. A benefit of using TuLiP in this case is that it is apparent if the specified cycles cannot be satisfied, because TuLiP then will generate an empty reactive module. In Supremica the specifications cannot express progress, so cycles cannot be forced by specifications. Since cycles cannot be forced, Supremica will synthesize a non-empty supervisor without cycles even if cycles are desired.

Paper C outlines a method to, in a sense, force cycles even in Supremica. To overcome the lack of progress properties in SCT, the plant is structured in a way such that a supervisor cannot prevent further execution cycles. This is achieved by having one marked location with an uncontrollable event on the outgoing transition. So, if the plant fires the uncontrollable event (which the supervisor cannot prevent), then the supervisor also has to allow the plant to fire the remaining events to complete the cycle (because of the marking). Although this technique does not produce infinite runs or progress, it makes sure that, if the plant so chooses, there is always the possibility for another cycle.

One reason to force cycles is to emulate a game, where two or more players take turns in a sequential manner. TuLiP synthesizes a reactive module by solving a game between the environment and the reactive module, which is a benefit in some applications. For instance, in the second case in Paper C, both cars must be allowed to decide their individual speed in each cycle (time step). The environment model or formal specification in TuLiP do not need formulae expressing how the speed updates shall be interleaved. Supremica, on the other hand, needs a separate plant whose only purpose is to ensure that the environment and supervisor update the speed in turns. Since these cycles represent

the passing of time in some sense, it is crucial that they can continue forever.

CHAPTER 5

Conclusions

Formal methods have several benefits as tools to design tactical planners for automated cars. Given the requirements and an environment model, the planner is proven to be correct. This provable correctness is in contrast to other methods, such as reviews and tests which can only find errors but cannot prove absence of errors. Furthermore, tactical planners make discrete decisions, and formal methods work on discrete states, so the requirements on tactical planners are natural to express in formal models. Tactical planners also act in dynamical feedback systems where they interact with the environment by actions and observations, which are built-in features of formal methods. An important capability of the formal methods in this regard is that they are capable of expressing temporal properties that detail how these interaction will or must continue over time.

Formal synthesis is an especially interesting class of formal methods because they can automatically generate the planner based on requirements and models. Formal synthesis removes the need to manually develop and implement the planner, so the development efforts can be directed to formalizing good requirements and good assumptions on the environment.

Demonstrations show that formal methods can be used to synthesize useful correct-by-construction tactical planners for automated cars. However, the application of formal synthesis to the generation of such planners has obstacles. These obstacles are mainly a problem of inspection and abstraction.

The first obstacle is that the output of formal synthesis methods are a black box that makes decisions based on previous inputs. Theoretically, the behavior of the black box is known since the model of the environment and the requirements are known. Practically, on the other hand, it is difficult to completely understand all possible interactions and

edge cases that result from the synthesis. One useful method to ascertain that the synthesis result is reasonably correct is to visually inspect it as an automaton, but this is only possible for planners with a small number of states. Another method is to simulate the tactical planner and check that it makes the desired decisions. However, using visual inspection or simulations to validate the requirements are no more effective than reviews and testing, which are the activities that formal methods were to amend. Consequently, if formal synthesis shall be used to generate tactical planners with hundreds of states, then the validation of the requirements requires consideration. Possible approaches to simplify the validation might be to find abstractions of synthesized planners, or build libraries of common models and requirements that are known to be correct.

The second obstacle concerns the level of abstraction of the tactical planners. Safe planners must avoid collisions, so the capability to express such requirements are crucial. Two cars have collided if they occupy the same physical space, so a requirement expressing 'do not collide' must be able to refer to the positions of the two cars. Moreover, to be practical in traffic, the resolution of the positions need to be high. As an example, if a road would be divided into 100 meter sections, then the planner cannot drive closer than 100 meters to any other car. The feasible resolution for a planner would rather be counted in single meters. Modeling all roads that the planner should operate on in such detail is infeasible. The solution is to model in a higher level of abstraction and use generic states that model modes of operation rather than, for instance, position. However, with a high level of abstraction it is not possible to express the same detailed requirements. As both detailed requirements and generic planners are desired, it seems valuable to investigate how the two can be combined.

5.1 Future Works

One possible drawback of SCT is that the synthesized supervisor restricts the system by disabling events rather than making decisions. Depending on the problem, the supervisor will have states from which more than one event is allowed because of the maximal permissiveness of the supervisor. If one or more events from such a state are controllable, it follows that some entity needs to make a decision whether to fire a certain event or to wait for an uncontrollable event to fire. That entity is called an event generator. The need for an event generator might limit the usefulness of SCT, especially if the event generator instead can be synthesized by RS. However, if the decisions to be made by an automated car can be divided into comfort decisions and safety decisions, then the possibility to have a separate event generator for comfort decisions and a supervisor to restrict it to safe decisions could be a versatile combination. The event generator could be generated with machine learning such that it generates events that are comfortable and aware of interactions, while the formal specification of the supervisor can focus solely on safety [58], [59].

CHAPTER 6

Summary of Included Papers

Paper A An earlier research study [16] employed formal verification and found issues in manually implemented code for an automotive application. Paper A explores whether these issues may be patched by using formal synthesis. Models and requirements are available, but the results show that it is not possible to use those directly to synthesize a meaningful patch. However, the attempted process of patching gives insights into prerequisites and limitations of synthesis.

Paper B In Paper B, an automated vehicle is tasked with a transport mission where safety is a high concern. Paper B employs formal verification to aid the design of a safety-critical tactical planner for the automated vehicle. The planner has requirements on temporal properties, which can be difficult and expensive to assure by testing and/or reviewing. The results indicate that formal methods can indeed be beneficial in automotive development processes. Continuous formal verification of the design and implementation while they are being developed means that 'bad' requirements and solutions are found early; simulations and real-world experiments of the complete system were performed successfully with little modification. The results also show that formal verification has the capability of proving functionality of a generic planner on a more specific environment model.

Paper C TuLiP and Supremica are tools from the fields of Reactive Synthesis and Supervisory Control Theory. Paper C compares the two synthesis methods implemented in TuLiP and Supremica from a modeling perspective. The comparison is based on two case studies; a turn-based game and an automated vehicle

scenario. Paper C demonstrates differences and similarities between the two synthesis tools, which can indicate in what situations a practitioner should use one before the other. For instance, it is demonstrated how shared resources affect the modeling, how cycles can be modeled, and how progress requirements are treated.

Paper D The tactical planner designed in Paper B was formally verified to be safe. However, formal verification has some drawbacks, so Paper D replicates the tactical planner in Paper B with the two different synthesis methods implemented in TuLiP and Supremica in order to investigate whether formal synthesis can alleviate the drawbacks. The synthesis methods are compared to each other, and to formal verification as performed in Paper B. The comparisons treat modeling, requirements, and the resulting planners. It is shown that the synthesized planners are similar to each other and that the requirements can be formalized in both of the synthesis methods' formalisms. Paper D gives insights into benefits and drawbacks of using formal synthesis to design tactical planners. As an example, it is difficult to inspect and interpret the results of synthesis, so it is sometimes difficult to know what behaviors interacting requirements are causing. Additionally, there is a conflict between the possibility to express detailed requirements while also generating generic planners.

References

[1] M. Hahm, "'Age in place': How self-driving cars will transform retirement", *Yahoo Finance*, Sep. 7, 2017.

[2] S. Szymkowski, "Lyft, Aptiv will supply rides to blind or visually impaired people", *The Car Connection*, Jul. 13, 2019.

[3] Intelligent Transport, "U.S. partnership brings self-driving technology to the blind", *Intelligent Transport*, Jul. 11, 2019.

[4] ITV, "Driverless cars tested by blind veterans", *itv*, Jul. 13, 2019.

[5] S. Abuelsamid, "Transition to autonomous cars will take longer than you think, Waymo CEO tells governors", *Forbes*, Jul. 20, 2018.

[6] J. Ramsey, "Nine driver assistance systems to become mandatory in Europe in 2022", *Auto Blog*, Mar. 26, 2019.

[7] SAE, *J3016 taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles*, Sep. 2016.

[8] N. Kalra and S. M. Paddock, *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, 2016.

[9] J. A. Michon, "A critical view of driver behavior models: What do we know, what should we do?", in *Human Behavior and Traffic Safety*, L. Evans and R. C. Schwing, Eds. Boston, MA: Springer US, 1985, pp. 485–524, ISBN: 978-1-4613-2173-6.

[10] P. Koopman, A. Kane, and J. Black, "Credible autonomy safety argumentation", in *Safety-Critical Systems Symposium*, Bristol UK, Feb. 2019.

[11] ISO/TC 22/SC 32, "ISO 26262: Road vehicles – functional safety", International Organization for Standardization, Tech. Rep., 2012.

[12] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008, ISBN: 9780262026499.

[13] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems, 2nd Edition.* Springer, 2010, ISBN: 0387333320;

[14] O. Kupferman, P. Madhusudan, P. Thiagarajan, and M. Vardi, “Open systems in reactive environments: Control and synthesis”, in *CONCUR 2000 — Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 1877, Springer, Berlin, Heidelberg, 2000.

[15] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, “How Amazon Web Services uses formal methods”, *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015, ISSN: 0001-0782.

[16] A. Zita, S. Mohajerani, and M. Fabian, “Application of formal verification to the lane change module of an autonomous vehicle”, in *13th IEEE Conference on Automation Science and Engineering (CASE)*, Aug. 2017, pp. 932–937.

[17] A. Brahmi, D. Delmas, M. H. Essoussi, F. Randimbivololona, A. Atki, and T. Marie, “Formalise to automate: Deployment of a safe and cost-efficient process for avionics software”, in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, Jan. 2018.

[18] Waymo. (2017). Waymo safety report - on the road to fully self-driving, [Online]. Available: `https://waymo.com/safetyreport/` (visited on 2019-12-10).

[19] Tesla. (2019). Tesla vehicle safety report, [Online]. Available: `https://www.tesla.com/VehicleSafetyReport` (visited on 2019-12-10).

[20] General Motors. (2019). General Motors self-driving safety report, [Online]. Available: `http://www.gm.com/mol/selfdriving.html` (visited on 2019-12-10).

[21] Uber Advanced Technologies Group. (2018). A principled approach to safety, [Online]. Available: `https://uber.app.box.com/v/UberATGSafetyReport` (visited on 2019-12-10).

[22] P. LeBeau, “Waymo hits 10 millionth mile, prepares for public ride hailing”, *CNBC*, Oct. 10, 2018.

[23] @Tesla. (2018). As of today Tesla owners have driven 1 billion(!) miles with autopilot engaged, [Online]. Available: `https://twitter.com/Tesla/status/1067810392322109441` (visited on 2019-12-10).

[24] I. Khrennikov, “Russia’s Yandex joins the self-driving car million-mile club”, *Bloomberg*, Oct. 17, 2019.

[25] D. Silver, “Miles driven”, *Medium*, Jan. 8, 2018.

[26] K. Wiggers, “Uber’s 250 autonomous cars have driven ‘millions’ of miles and transported ‘tens of thousands’ of passengers”, *VentureBeat*, Apr. 11, 2019.

[27] B. Vlasic and N. E. Boudette, “Self-driving Tesla was involved in fatal crash, U.S. says”, *The New York Times*, Jun. 30, 2016.

[28] D. Wakabayashi, “Self-driving Uber car kills pedestrian in Arizona, where robots roam”, *The New York Times*, Mar. 19, 2018.

[29] Q. C. S. Corp., "NHTSA's implausible safety claim for Tesla's autosteer driver assistance system", Quality Control Systems Corporation, Crownsville, Maryland, Tech. Rep., Feb. 8, 2019.

[30] The Tesla Team. (2019). Introducing software version 10.0, [Online]. Available: `https://www.tesla.com/blog/introducing-software-version-10-0` (visited on 2019-12-10).

[31] T. Louw, J. Kuo, R. Romano, V. Radhakrishnan, M. G. Lenné, and N. Merat, "Engaging in NDRTs affects drivers' responses and glance patterns after silent automation failures", *Transportation Research Part F: Traffic Psychology and Behaviour*, vol. 62, pp. 870–882, 2019, ISSN: 1369-8478.

[32] A. Pnueli, "The temporal logic of programs", in *18th Annual Symposium on Foundations of Computer Science*, Oct. 1977, pp. 46–57.

[33] S. Safra, "On the complexity of omega-automata", in *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, ser. SFCS '88, Washington, DC, USA: IEEE Computer Society, 1988, pp. 319–327, ISBN: 0-8186-0877-3.

[34] M. Y. Vardi and P. Wolper, "Automata-theoretic techniques for modal logics of programs", *Journal of Computer and System Sciences*, vol. 32, no. 2, pp. 183–221, 1986, ISSN: 0022-0000.

[35] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray, "Control design for hybrid systems with TuLiP: The temporal logic planning toolbox", in *IEEE Conference on Control Applications (CCA)*, Sep. 2016, pp. 1030–1041.

[36] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica – an efficient tool for large-scale discrete event systems", *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5794–5799, 2017, 20th IFAC World Congress, ISSN: 2405-8963.

[37] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, First. Addison-Wesley Professional, 2003, ISBN: 0-321-22862-6.

[38] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer, "How to handle assumptions in synthesis", in *Proceedings 3rd Workshop on Synthesis*, K. Chatterjee, R. Ehlers, and S. Jha, Eds., ser. Electronic Proceedings in Theoretical Computer Science, vol. 157, Vienna, Austria: Open Publishing Association, Jul. 2014, pp. 34–50.

[39] P. Madhusudan, "Control and synthesis of open reactive systems", PhD thesis, University of Madras, 2001.

[40] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning", *IEEE Transactions on Automatic Control*, vol. 57, no. 11, pp. 2817–2830, 2012.

[41] B. Finkbeiner, "Synthesis of reactive systems", in *Dependable Software Systems Engineering*, vol. 45, 2016, pp. 72–98.

[42] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. Murray, "TuLiP: A software toolbox for receding horizon temporal logic planning", in *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, Apr. 2011, pp. 313–314.

[43] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs", in *Verification, Model Checking, and Abstract Interpretation*, E. Emerson and K. Namjoshi, Eds., ser. Lecture Notes in Computer Science, vol. 3855, Springer, Jan. 2006, pp. 364–380.

[44] U. Klein and A. Pnueli, "Revisiting synthesis of GR(1) specifications", in *Hardware and Software: Verification and Testing*, S. Barner, I. Harris, D. Kroening, and O. Raz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 161–181, ISBN: 978-3-642-19583-9.

[45] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems", *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[46] T. Moor, "Supervisory control of non-terminating processes: An interpretation of liveness properties", Lehrstruhl für Regelungstechnik, Friedrich-Alexander Universität Erlangen-Nürnberg, Tech. Rep., Nov. 25, 2017.

[47] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model", in *Proceedings of the 30th International Design Automation Conference*, ser. DAC '93, Dallas, Texas, USA: ACM, 1993, pp. 86–91, ISBN: 0-89791-577-1.

[48] M. Sköldstam, K. Åkesson, and M. Fabian, "Modeling of discrete event systems using automata with variables", in *Proceedings of the 46th IEEE Conference on Decision and Control*, 2007, pp. 3387–3392, ISBN: 1-4244-1498-9.

[49] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams", *ACM Computing Surveys (CSUR*, vol. 24, no. 3, pp. 293–318, Sep. 1992, ISSN: 0360-0300.

[50] Z. Fei, S. Miremadi, K. Åkesson, and B. Lennartson, "Efficient symbolic supervisor synthesis for extended finite automata", *IEEE Transactions on Control Systems Technology*, vol. 22, pp. 2368–2375, 6 2014, ISSN: 1063-6536.

[51] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Synthesis of control protocols for autonomous systems", *Unmanned Systems*, vol. 01, no. 01, pp. 21–39, 2013.

[52] F. F. Reijnen, M. A. Goorden, J. M. Van De Mortel-Fronczak, and J. E. Rooda, "Supervisory control synthesis for a waterway lock", in *1st Annual IEEE Conference on Control Technology and Applications, CCTA 2017*, Aug. 2017, pp. 1562–1568, ISBN: 9781509021826.

[53] T. Korssen, V. Dolk, J. Van De Mortel-Fronczak, M. Reniers, and M. Heemels, "Systematic model-based design and implementation of supervisors for advanced driver assistance systems", *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 2, pp. 533–544, Feb. 2018, ISSN: 15249050.

[54] D. Gunning and D. Aha, "DARPA's explainable artificial intelligence (xai) program", *AI Magazine*, vol. 40, no. 2, pp. 44–58, Jun. 2019.

[55] R. Schmelzer, "Understanding explainable AI", *Forbes*, Jul. 23, 2019.

[56] R. Kumar and V. K. Garg, "Optimal supervisory control of discrete event dynamical systems", *SIAM Journal on Control and Optimization*, vol. 33, no. 2, pp. 419–439, 1995.

[57] S. Ware and R. Malik, "Supervisory control with progressive events", in *11th IEEE International Conference on Control Automation (ICCA)*, Jun. 2014, pp. 1466–1471.

[58] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding", in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018, pp. 2669–2678.

[59] D. Liu, M. Brännström, A. Backhouse, and L. Svensson, "Learning faster to perform autonomous lane changes by constructing maneuvers from shielded semantic actions", in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, Oct. 2019, pp. 1838–1844.