THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Formal Methods and Safety for Automated Vehicles

Modeling, Abstractions, and Synthesis of Tactical Planners

Jonas Krook

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Formal Methods and Safety for Automated Vehicles

Modeling, Abstractions, and Synthesis of Tactical Planners

Jonas Krook

ISBN 978-91-7905-731-2

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5197
ISSN 0346-718X

Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31 772 1000

Cover:
A partition of the state space of a double integrator. See Fig. 5.2 on page 67.

Printed by Chalmers Reproservice
Gothenburg, Sweden, October 2022

Abstract

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

Tactical planners are responsible for making discrete decisions for the coming seconds or minutes. As with all subsystems in an automated vehicle, these planners need to be supported with a credible and convincing argument of their correctness. The planners interact with other road users in a feedback loop, so their correctness depends on their behavior in relation to other drivers and road users over time. One way to ascertain their correctness is to test the vehicles in real traffic. But to be sufficiently certain that a tactical planner is safe, it has to be tested on 255 million miles with no accidents.

Formal methods can, in contrast to testing, mathematically prove that given requirements are fulfilled. Hence, these methods are a promising alternative for making credible arguments for tactical planners' correctness. The topic of this thesis is the use of formal methods in the automotive industry to design safe tactical planners. What is interesting is both how automotive systems can 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 viable to formally express desired properties of tactical planners, and to use formal methods to prove their correctness. However, the difficulty to anticipate and inspect the interaction of several desired properties is found to be an obstacle. Model Checking, Reactive Synthesis, and Supervisory Control Theory have been used in the design and development process of tactical planners, and these methods have their benefits, depending on the application. To be feasible and useful, these methods need to operate on both a high and a low level of abstraction, and this thesis contributes an automatic abstraction method that bridges this divide.

It is also found that artifacts from formal methods tools may be used to convincingly argue that a realization of a tactical planner is safe, and that such an argument puts formal requirements on the vehicle's other subsystems and its surroundings.

Keywords: Formal methods, safety case, automated vehicles, tactical planning, formal verification, formal synthesis, model checking, reactive synthesis, supervisory control theory, automatic abstraction.

List of Publications

This thesis is based on the following publications:

Paper A **Jonas Krook**, Yuvaraj Selvaraj, Wolfgang Ahrendt, Martin Fabian, "A Formal-Methods Approach to Provide Evidence in Automated-Driving Safety Cases," submitted to *IEEE Transactions on Intelligent Vehicles*, 2022.

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

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

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

Paper E **Jonas Krook**, Roozbeh Kianfar, Martin Fabian, "Formal Synthesis of Safe Stop Tactical Planners for an Automated Vehicle," *IFAC-PapersOnLine*, 2020, 15th IFAC Workshop on Discrete Event Systems (WoDES).

Paper F **Jonas Krook**, Robi Malik, Sahar Mohajerani, Martin Fabian, "Robust Stutter Bisimulation for Abstraction and Controller Synthesis with Disturbance," submitted to *Automatica*, 2022.

Other publications by the author, not included in this thesis, are:

Paper G Yuvaraj Selvaraj, **Jonas Krook**, Wolfgang Ahrendt, Martin Fabian, "On How to Not Prove Faulty Controllers Safe in Differential Dynamic Logic," in *23rd International Conference on Formal Engineering Methods*, Oct. 2022.

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 while I was writing this thesis.

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

My collaborators Yuvaraj Selvaraj, Robi Malik, and Wolfgang Ahrendt have all, each on their own, made a substantial impression on me and the works covered by this thesis. Thank you for your contributions!

I also want to send a big thank you to my group at Zenseact who have supported and encouraged me, without whom life as a PhD student would have been less fun and interesting.

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, October 2022

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

Acronyms

AV:	Automated Vehicle
ADS:	Automated Driving System
ASIL:	Automotive Safety Integrity Level
BDD:	Binary Decision Diagram
DDT:	Dynamic Driving Task
DES:	Discrete Event System
E/E:	Electrical or Electronic
FuSa:	Functional Safety
GR(1):	General Reactivity of Rank 1
LSM:	Lateral State Manager
LTL:	Linear Temporal Logic
MC:	Model Checking
MRC:	Minimal Risk Condition
MRM:	Minimal Risk Maneuver
OEDR:	Object and Event Detection and Response
ODD:	Operational Design Domain
RS:	Reactive Synthesis
SOTIF:	Safety of the Intended Function
SCT:	Supervisory Control Theory

Contents

Part I

Overview

CHAPTER 1

Introduction

Automated Vehicles (AVs) are road vehicles that under certain periods of time can perform the driving task without continuous supervision by a driver [1]. The features that operate an AV are referred to as *Automated Driving Systems* (ADS) [2]. These systems are set apart from both active safety systems and driver support systems in that active safety systems are activated to avoid accidents, whereas driver support systems operate under the supervision of the driver. Thus, none of the latter systems perform the actual driving task. In contrast, an ADS completely relieve people from the driving tasks in a certain traffic environment, and hence may enable vehicle usage that is not possible with supervised features or during manual driving. Also, removing the human driver has the potential to reduce the effects of human weaknesses on driving and traffic.

On the far end of the unsupervised spectrum are the Fully Automated Vehicles, which are equipped with features that perform the driving tasks without any supervision from a driver [1]. These features completely free people from driving, and the introduction of such vehicles is thought to bring several benefits and new application areas. One example is that people who currently cannot or should not drive, for instance elderly or visually impaired

people, might get access to better personal mobility with the introduction of fully automated vehicles [3]–[6].

Even if not fully automated, there are several ways in which AVs may be beneficial. For instance, it is likely that the inter-vehicle distance in traffic can be reduced, and that AVs can help reduce traffic jams [7]. Experience has also shown that the fuel economy can be improved by driver support systems, and this benefit will likely carry over to AVs [7]. These effects all have the potential to reduce emissions and infrastructure need.

The relief from the driving task that AVs bring also allows the driver to engage in other tasks. For instance, some people might want to engage in leisure activities in the vehicle instead of manually driving in congested traffic [8]. However, a drawback might be that such benefits for the individual driver may increase traveling by car, and by that effect pose a challenge for the sustainability of the traffic system.

Another important benefit of AVs is believed to be improved traffic safety. Many traffic accidents are attributed to human error [1], [7], and past experience of active safety and driver support systems has shown that their introduction significantly reduces the prevalence of certain accident types [7]. Thus, it is not unlikely that removing the driver and replacing them with more advanced features further reduces accidents. Although the share of accidents that are actually caused by human error might often be overrated [9]–[11], AVs still seem to have the opportunity to decrease the number of traffic fatalities. This decrease is thought to be possible by improving on active safety and driver support systems as well as by increasingly weakening the dependence on the driver [7], [12].

Naturally, if traffic safety is to increase by automation, then any driving automation system that performs some or all of the driver's driving tasks must be safer than a human driver. This applies to traffic situations where drivers perform poorly, but also where they already perform well; both when it comes to handling a wide variety of situations and in cases where the vehicle suffers from malfunctions. Although human drivers cause a lot of traffic accidents, they still have a very low accident and failure rate [13]. This low rate poses difficulties when AVs are developed; if these vehicles are to 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 safe AVs must be. However, this does

not mean that all AVs have the same requirements on safety. The more driving tasks and situations that are to be handled by an AV, the more work goes into ensuring and proving that the system is sufficiently safe, and, consequently, there might be a long time until Fully Automated Vehicles are available [14].

The safety of the AV must be supported by a credible and convincing argument of its correctness [15], and one method to ascertain this can be by real world driving. But then, the vast amount of data needed [13] and the fact that failures during the driving will add millions of kilometers to the needed total mileage [15] means that complete-vehicle testing is implausible and scales poorly to accommodate updates of the system. The data need can be reduced by using other statistical methods to estimate the failure rate from the distribution of a threat metric [16], but it might still be prohibitively costly to apply a corrective update to the system and redo the testing.

The ADS is typically realized in several electrical or electronic (E/E) subsystems, each with a specific sub-task that can be verified independently of the others. By verifying the subsystems independently before complete-vehicle testing, the risk of faults or deficiencies during the complete-vehicle testing is reduced, and so is the required milage. For the purpose of this thesis the AV'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 estimate of the environment. Based on the fused sensor information, the *Decision and Control* subsystem 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.

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 [17], as 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 minutes, 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 millisec-

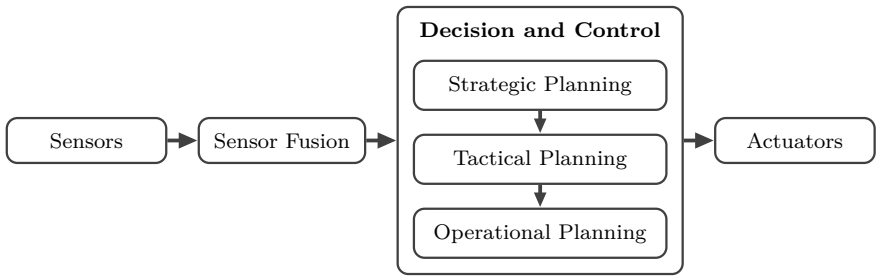

Figure 1.1: System overview.

onds up to a second, and examples are speed control and sudden avoidance maneuvers.

If the E/E subsystems are to support the correctness argument of the complete system, each of the four subsystems in Figure 1.1 must be supported by a credible and convincing argument of their correctness [15]. Since the subsystems have different tasks and solve them with different approaches their correctness is 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 [18]. This is a field with mature analytical methods developed from the 1930's and onward. The sensors and sensor fusion subsystems can use statistical approaches to verify their correctness in open-loop simulations [19]. Given a recorded input, an accuracy of the estimates compared to ground-truth data can be calculated. The main issue for the sensors and Sensor Fusion subsystems is that a lot of data is needed [13], and that reliable ground-truth data or annotations can be laborious to attain [20].

Many of the analytical tools from control theory are not suitable for analysis of 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 in general be used on discrete-state systems. The open-loop simulations that are useful for the perception 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 [13]. This thesis explores ways to limit the needed quantitative and statistical verification effort of tactical planners, and tries to focus on achieving provably correct planners as a means to acquire credible and convincing arguments of the correctness of their construction.

Two standard methods to provide evidence that fault and deficiency prevalence in E/E subsystems are sufficiently low are *reviewing* and *testing* [21]. 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 in large parts of the development process [21], but neither of them can provide proofs of correctness; they can only find faults, not show absence of them [22]. Also, when the design or implementation of a part is changed the reviewing and testing must be performed again.

An alternative approach is *formal methods* [22], [23], which 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 with well defined and unambiguous semantics, which means that they are suitable for automatic machine reasoning. The formal language is used to formalize the requirements into a specification, and to construct a formal model. After the specification and the model are available 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 [22]. This model can be the actual software code, a simplified abstraction of it, or a simple model of the intended functionality of the item; this depends on what requirements are being verified. When the specification and the formal model are available, a formal verification tool can search for counterexamples. If none are found, then the model fulfills the specification. Because formal verification comes with a proof or a counterexample, less effort has to be spent on reviews and tests. Since much of the process is automated, it can be run again and again without much manual effort, further easing the effort compared to 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 [23], [24]. Based on the specification and the formal model, formal synthesis automatically computes a correct-by-construction implementation. This has the advantage over formal verification that manual implementation, often and error prone and time consuming process, can be avoided. Also, changes in the requirements does not necessarily incur any manual re-implementation.

Formal methods, especially formal verification, have been used successfully in industrial applications [25]–[27]. 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 program 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 overall purpose of this thesis is to investigate *whether 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.

This thesis attempts to evaluate how problems in the development of tactical planners in AVs 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 AVs.

1.1 Hypothesis

The hypothesis of this thesis starts with 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 as 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, do not seem to be widely adopted for generation of tactical planners

for AVs, probably there 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. The assumption is that coarse abstractions that take planner capabilities into account may contribute to less effort being allowed to be spent on justifying the safety of the formally proven tactical planners.

1.2 Research questions

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

RQ 1 What are the current limitations of formal synthesis for tactical planners for automated vehicles that hinders its adoption in the automotive industry?

RQ 2 To what problems regarding tactical planners can synthesis and verification be applied?

RQ 3 What level of abstraction is possible to use for tactical planners for automated vehicles, and will that level facilitate meaningful and non-trivial planners?

1.3 Contributions

The main contributions of this thesis are:

- It presents how formal methods may be used to provide evidence that tactical planners fulfill their safety requirements in the context of applicable standards. This is both a contribution to answer RQ 1, but also an attempt to lessen the limitations of formal synthesis (Paper A).
- It identifies the difficulty to inspect the result of synthesis as an obstacle to adoption of formal synthesis, providing one piece in the answer of RQ 1 (Paper D, Paper C, Paper E).

- It identifies a conflict between the desire to express detailed requirements and to generate generic tactical planners, which gives some answers to RQ 3 (Paper B, Paper E).

- It provides insights into how verification and synthesis models differ when it comes to separation of environment, requirements, and planner, which helps in answering which method to use in relation to RQ 2 (Paper B, Paper E).

- 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, answering which synthesis methods should be chosen in respect to RQ 2 (Paper C, Paper E).

- It presents the *robust stutter bisimulation* relation that can be used to automatically create coarse abstractions for non-deterministic systems. The relation is useful to reduce the size of the state space for feedback systems such as ADS, where the environment includes actions taken by other road users, providing a partial answer to RQ 3 (Paper F).

1.4 Method

The research of this thesis has mainly been exploratory and qualitative. As such, the work has been uncovering and formulating new research problems and questions, as much as it has been trying to answer the above questions and accept or reject the hypothesis.

One part in answering all research questions and confirming the hypothesis is to investigate how to construct a convincing safety argument based on formal methods. Paper A provides results to this mean by employing an argumentative approach to contribute to the research in this area. This method was chosen because the safety research on the use of formal methods applied to tactical planners still seems to be in its infancy. Furthermore, the safety research on automated vehicles overall is still debating which methods are best suited to ensure safety. Therefore, it could be valuable to broaden this debate with approaches on using formal methods to argue for safety. A complete

study on approaches to verify and validate this approach would likely require large amounts of data and an already implemented system that is evaluated in real-world road traffic. Such an extensive study would be infeasible in the scope of this thesis.

Focus has been on exploring the technical aspects of the research questions and hypotheses. Hence, Paper B, Paper C, Paper D, and Paper E evaluate different aspects of formal methods in automotive safety-critical development by various case studies. These case studies focus more on the limitations of the tools themselves, and less on evaluating limitations arising from organizational perspectives.

Paper B applies formal verification to ensure that a developed tactical planer for an automated vehicle always reaches a safe state. This particular case study was chosen because it is a pertinent problem for Automated Vehicles. The study created a model of the traffic environment in which relevant and realistic requirements were proven. Then the model was validated both by simulations and real world tests to assess the feasibility of using formal verification for that class of problems.

Paper C compares several aspects of the two different formal synthesis paradigms *Reactive Synthesis* and *Supervisory Control Theory* by applying each method to two different sample problems. These two paradigms were chosen because several aspects of the paradigms have different characteristics, and the sample problems were likewise chosen such that a variety of types of requirements and dynamics were present. The comparison evaluates different qualitative properties that are useful for choosing a specific formal method for different problems.

Formal verification previously found problems in an implementation of a planner for lane changes, and Paper D further explored how those problems could be fixed by using synthesis. As in Paper C, the intent was to qualitatively investigate useful properties of formal synthesis, but here more specifically in the automotive domain.

The fourth case study was conducted in Paper E. The same setting as in Paper B was used to compare the same formal methods as in Paper C, but now in an automated-driving setting.

For these four case studies, realistic requirements and scenarios have been formulated, and formal methods have been used to design correct-by-construction tactical planners. These planners have then been evaluated by simula-

tions, physical experiments, or visual inspections.

Some results from the case studies implied that abstraction techniques are important to ease the use of formal methods in the automotive domain. Paper F presents one such technique, and shows that it is relevant for Reactive Synthesis. The relevance is established by the use of mathematical analysis and a small proof of concept.

1.5 Outline

This thesis is divided into two parts. Part I introduces safety of AVs, explains some fundamentals of formal methods, and puts the appended papers into context. Part II contains the appended papers forming the basis of the thesis.

Part I consists of the following chapters: Chapter 2 goes through the fundamental concepts of formal methods that are used in later chapters; Chapter 3 presents the problem of credibly arguing for the safety of AVs; considerations, benefits, and obstacles when using formal methods to develop correct-by-construction tactical planners are treated in Chapter 6; modeling and automatic abstraction are treated in Chapter 5; Chapter 6 concludes the findings in this thesis and presents future challenges; finally, Chapter 7 gives short summaries of the appended papers.

CHAPTER 2

Formal Methods

Formal Methods is a class of mathematically rigorous techniques and tools that are used to specify, design, verify, and synthesize components; mainly by allowing rigorous reasoning about correctness of these components [28]. At the foundation of these methods are formal languages, which provide rigor by establishing precise and unambiguous definitions of language syntax and semantics. Every statement of such a language is thus well-formed and has a precise meaning. Many formal languages used at the core of formal methods allow automatic or semi-automatic generation of proofs for statements, where the interesting statements to prove typically can be the formalized requirements of some subsystem in Fig. 1.1.

Examples of formal languages are different automata and logic based languages, where, for instance, propositional logic and regular expressions are two concrete examples of languages with formal syntax and semantics. In this thesis the interest lies mainly in formal systems whose formalism allows to be interpreted over time, since this allows reasoning about a vehicle's behavior over time.

2.1 Regular languages

A formal language is defined as a set of *words* constructed from a set of *letters* Σ, the *alphabet*, according to a set of rules. *Regular* languages are a specific subset of the formal languages.

Regular languages consist of words that are sequences of the letters of an alphabet Σ. The word of zero length of the alphabet Σ is denoted by $\Sigma^0 = \{\varepsilon\}$, where ε is the *empty word*. The words of length $n+1$ are defined inductively as $\Sigma^{n+1} = \{\rho\, a \mid \rho \in \Sigma^n \text{ and } a \in \Sigma\}$, where $\rho\, a$ denotes the sequence acquired from concatenation of the word ρ with the letter a. The *Kleene closure* of an alphabet Σ, denoted by Σ^*, is the language of all the finite words constructed from Σ, i.e., $\Sigma^* = \bigcup_{n\in\mathbb{N}} \Sigma^n$. The language of infinite words over Σ is denoted by Σ^ω, and the language of all finite and infinite words over Σ is $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. The concatenation of two languages L_1 and L_2 is $L_1 L_2 = \{\rho_1\rho_2 \mid \rho_1 \in L_1 \text{ and } \rho_2 \in L_2\}$. A finite word $\rho \in \Sigma^*$ is a *prefix* of the word $\pi \in \Sigma^\infty$, written $\rho \sqsubseteq \pi$, if there exists a word $\pi' \in \Sigma^\infty$ such that $\rho\pi' = \pi$. The word $\rho \in \Sigma^*$ is a *proper prefix*, written $\rho \sqsubset \pi$, if $\rho \sqsubseteq \pi$ and $\rho \neq \pi$.

A language L is $*$*-regular* (or simply *regular*) if all the words of the language are built by finite number of concatenations of letters in the alphabet Σ [29], and a language L is ω*-regular* if all the words of the language are built by infinite number of concatenations of the letters in the alphabet Σ [22].

The set of finite prefixes of a word $\pi \in \Sigma^\infty$ is $\mathsf{pfx}(\pi) = \{\rho \in \Sigma^* \mid \rho \sqsubseteq \pi\}$. The *closure* of a set of finite words $L \subseteq \Sigma^*$ is the set of words whose prefixes are all in L, defined by $\mathsf{clo}(L) = \{\pi \in \Sigma^\infty \mid \mathsf{pfx}(\pi) \subseteq L\}$. Hence, Σ^ω is the closure of Σ^*, i.e., $\Sigma^\omega = \mathsf{clo}(\Sigma^*)$.

The regular and ω-regular languages are closely connected to transition systems, which can be used to model regular and ω-regular languages.

2.2 Transition Systems

Transition systems consist of states and transitions [22]. The transitions between the states are labeled with transition labels, that define which inputs a certain state can consume and how the transition system transits to the next state given that input. The states are labeled with *atomic propositions* by a labeling function, which makes it possible to refer to specific states, or groups of states, by their propositions. Transition systems and derivatives thereof

form the foundation of most of the papers and methods considered in this thesis.

Definition 1: *A* transition system *(TS) is a tuple*

$$TS = \langle S, \Sigma, \delta, S^\circ, \mathrm{AP}, L \rangle \tag{2.1}$$

where

- S *is a set of* states*;*
- Σ *is a set of* transition labels*;*
- $\delta \subseteq S \times \Sigma \times S$ *is a* transition relation*;*
- $S^\circ \subseteq S$ *is a set of* initial states*;*
- AP *is a set of* atomic propositions*;*
- $L \colon S \to 2^{\mathrm{AP}}$ *is a* labeling function.

If the set of states S is finite, then TS is a finite transition system*, and if the transition relation δ is a function, then TS is a* deterministic transition system*.*

When a transition system changes state from one state to the next, it takes a *step* along a transition. A sequence of steps through a transition system can be seen as progress of time, either by requiring that a step from the current state to the next state at certain time instances, or by considering the transition labels as events that fire at arbitrary times and makes the system step to the next state. Thus, the traversals of the states of a transition system can be the basis for reasoning about temporal properties.

A finite sequence of states $\rho = s_1 \cdots s_n \in S^*$ is a *finite path fragment* of a transition system TS if there exists a sequence of transition labels $w = \sigma_1 \cdots \sigma_{n-1} \in \Sigma^*$ such that $(s_i, \sigma_i, s_{i+1}) \in \delta$ for all $1 \leq i < n$. The set of all finite path fragments of a transition system TS is denoted by $\mathrm{Frags}^*(TS)$. An infinite sequence of states $\pi = s_1 s_2 \cdots \in S^\omega$ is an *infinite path fragment* of TS if all prefixes are path fragments of TS, i.e., if $\mathsf{pfx}(\pi) \subseteq \mathrm{Frags}^*(TS)$. The set of all infinite path fragments, and the set of all path fragments of a transition system TS are denoted by $\mathrm{Frags}^\omega(TS)$ and $\mathrm{Frags}^\infty(TS)$, respectively. A path fragment $\pi = s_1 s_2 \cdots \in S^\infty$ is a *path* of a transition system TS if $s_1 \in S^\circ$, and the set of paths of a transition system TS is denoted by $\mathrm{Paths}^\omega(TS)$.

The labeling function L labels each state of the transition system TS with a subset of the atomic propositions AP with which individual states or groups of states can be referred to. In some instances, the distinction between propositions and states is important because the states are considered unobservable, whereas the labels are observable, i.e., an external entity does not know which state a transition system is in, but it knows the label of the current state.

A path fragment of a transition system leaves a *trace* of state labels. Formally, the trace of a finite path fragment is given by a function $\mathsf{trace}\colon S^* \to (2^{\mathrm{AP}})^*$, where its definition is given recursively for a finite path fragment $\rho = \rho' s_n \in S^*$ (with $s_n \in S$) by

$$\mathsf{trace}(\rho) = \mathsf{trace}(\rho' s_n) = \begin{cases} L(s_n) & \text{if } \rho' = \varepsilon \ ; \\ \mathsf{trace}(\rho') L(s_n) & \text{otherwise} \ . \end{cases} \tag{2.2}$$

The trace function can be extended to infinite path fragments as $\mathsf{trace}\colon S^\infty \to (2^{\mathrm{AP}})^\infty$, where the output is an infinite sequence of subsets of AP.

The set of *trace fragments* of a state s of a transition system TS, written as $\mathsf{traces}(s)$, is the set of all traces of path fragments starting from the state s, i.e., $\mathsf{traces}(s) = \{\mathsf{trace}(\pi) \in (2^{\mathrm{AP}})^\infty \mid s \sqsubseteq \pi \in \mathrm{Frags}^\infty(TS)\}$. The set of *traces* of a transition system TS, denoted $\mathsf{traces}(TS)$, is the set of trace fragments of the initial states of TS, i.e., $\mathsf{traces}(TS) = \{\mathsf{traces}(s) \mid s \in S^\circ\}$.

Path fragments and traces thereof can contain repetitions of a state or a state label, respectively. In some cases these repetitions are not important, and it can therefore be beneficial to disregard them; it is sometimes irrelevant how long a system resides in a state or set of states. For instance, safety requirements for a vehicle might specify that crashes must be avoided, and that it must be possible to at all times reach a stopped position where the risk of harm is acceptably low. Both these requirements must be fulfilled, but there is no reference to specific time instances, so path fragments that only differ in the number of repetitions of each state or label can be considered equivalent.

A step $(s_i, \sigma, s_{i+1}) \in \delta$ of a transition system TS is a *stutter step* if $s_i = s_{i+1}$. A *stutter free* sequence $\mathsf{sf}(\pi) \in S^\infty$ is obtained from the path fragment $\pi = s_1 s_2 \cdots \in S^\infty$ by removing all elements s_{i+1} such that $s_{i+1} = s_i$. Stutter free traces are obtained similarly by checking whether $L(s_{i+1}) = L(s_i)$. Two path fragments π_1 and π_2 are *stutter equivalent* if they are equivalent modulo stuttering, i.e., if $\mathsf{sf}(\pi_1) = \mathsf{sf}(\pi_2)$ and if π_1 and π_2 are either both finite or both

infinite. Two path fragments π_1 and π_2 are *stutter trace equivalent* if they are equal when repetitions are removed, i.e., if $\mathsf{sf}(\mathsf{trace}(\pi_1)) = \mathsf{sf}(\mathsf{trace}(\pi_2))$ and if $\mathsf{trace}(\pi_1)$ and $\mathsf{trace}(\pi_2)$ are either both finite or both infinite.

Kripke Structures

A special case of transition systems are *Kripke structures*, which omit the transition labels Σ.

Definition 2: *A Kripke structure is a tuple*

$$K = \langle S, S^\circ, \delta, \mathrm{AP}, L \rangle,$$

where S is a set of states, $S^\circ \subseteq S$ is a set of initial states, $\delta \subseteq S \times S$ is a transition relation, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function that assigns a set of atomic propositions to each state.

Paths and traces are defined analogously for Kripke structures as for transition systems, except that the steps of a path fragment do not need a transition label to be taken, i.e., a sequence of states $\pi = s_1 s_2 \cdots \in S^\infty$ is a path fragment of a Kripke structure K as long as $(s_i, s_{i+1}) \in \delta$.

Finite State Automata

Automata are modifications of transition systems where the labeling function is removed and a set defining an *acceptance condition* is introduced.

Definition 3: *A* non-deterministic finite state automaton *(NFA) is a tuple*

$$A = \langle S, \Sigma, \delta, S^\circ, F \rangle \ ,$$

where S is a finite set of states, Σ is a finite set of transition labels, the alphabet, $S^\circ \subseteq S$ is a set of initial states, $\delta \subseteq S \times \Sigma \times S$ is a transition relation, and $F \subseteq S$ is a set representing the acceptance condition. If δ is a function and S° a single state, then the automaton is called a deterministic finite state automaton *(DFA).*

Let $w = \sigma_1 \sigma_2 \cdots \in \Sigma^\infty$ be a word. A *run* for w in A is a path $\rho = s_1\, s_n \cdots \in S^\infty$ such that $(s_i, \sigma_i, s_{i+1}) \in \delta$ for all $i \geq 1$. So a word of an NFA is formed by concatenating the letters on the transitions taken when stepping through a path of the NFA.

The acceptance condition of an NFA A is a set of states, i.e., $F \subseteq S$. A finite run $s_1 \cdots s_n \in S^*$ is called *accepting*, or *marked*, by A if the end state s_n

is in F. A word w of A is accepted, or marked, by A if there is an accepting run for w in A.

The *language* of an NFA A, denoted $\mathcal{L}(A)$, is the set of all finite words over the alphabet that has a run in A, i.e., $\mathcal{L}(A) = \{w \in \Sigma^* \mid \text{there exists a run for } w \text{ in } A\}$. Correspondingly, the *accepted language*, or *marked language*, of an NFA A, is the set of all accepted words of A. It can be shown that the collection of languages accepted by NFAs is exactly the collection of regular languages [22].

NFAs accept runs for which the end state is in the acceptor set, which means that accepted runs and accepted words are finite sequences. The accepting condition can be changed to construct automata whose marked languages consist of infinite words. These automata are called *ω-automata*. The structure of the acceptance condition F of ω-automata can have different structure, which leads to a rich plethora of automata with marked languages of infinite words. ω-automata will be treated below, but first an extension of NFAs is presented.

ω-Automata

ω-automata have the same structure as NFAs, but the acceptance condition only accepts infinite words. The set of states that occur infinitely many times in a run $\pi = s_0 s_1 \cdots \in S^\omega$ is obtained by the function $\text{Inf}(\pi) = \text{Inf}(s_0 s_1 \cdots) = \{s \in S \mid \text{there exists an infinite set } N \subseteq \mathbb{N} \text{ such that for all } i \in N, s = s_i\}$.

A *non-deterministic Büchi automaton* (NBA) BA is obtained from an NFA $A = \langle S, \Sigma, \delta, S^\circ, F\rangle$ by changing the acceptance condition such that an infinite run $\pi = s_1 s_2 \cdots \in S^\omega$ is accepted by BA if at least one of the states occurring infinitely often in π is in F, i.e., if $\text{Inf}(\pi) \cap F \neq \emptyset$. A word w of an NBA BA is accepted if there exists an accepted run for w in BA. If δ is a function and S° is a single state, then BA is a *deterministic Büchi automaton* (DBA).

It can be shown that the collection of languages accepted by NBAs is exactly the collection of ω-regular languages [22]. However, the collection of languages accepted by DBAs do not include all ω-regular languages, so DBAs are stricty less expressive than NBAs. This is in contrast to DFAs and DFAs, which are equally expressive.

Compositions of Transition Systems

Systems and specifications are often easier to formalize when they are decomposed and separate. However, the semantics of their combinations is needed to analyze their interactions.

Definition 4: *Given two NFA* $A_1 = \langle S_1, \Sigma_1, \delta_1, S_1^\circ, F_1 \rangle$ *and* $A_2 = \langle S_2, \Sigma_2, \delta_2, S_2^\circ, F_2 \rangle$*, the* full synchronous composition *of* A_1 *and* A_2 *is* $A_1 \parallel A_2 = \langle S_1 \times S_2, \Sigma_1 \cup \Sigma_2, \delta, S_1^\circ \times S_2^\circ, F_1 \times F_2 \rangle$*, where:*

- $((s_1, s_2), \sigma, (s_1', s_2')) \in \delta$ *if* $\sigma \in \Sigma_1 \cap \Sigma_2$, $(s_1, \sigma, s_1') \in \delta_1$, $(s_2, \sigma, s_2') \in \delta_2$,
- $((s_1, s_2), \sigma, (s_1', s_2')) \in \delta$ *if* $\sigma \in \Sigma_1 \setminus \Sigma_2$, $(s_1, \sigma, s_1') \in \delta_1$,
- $((s_1, s_2), \sigma, (s_1', s_2')) \in \delta$ *if* $\sigma \in \Sigma_2 \setminus \Sigma_1$, $(s_2, \sigma, s_2') \in \delta_2$.

The full synchronous composition of two NFA is itself an NFA. The state set of a full synchronous composition is all the pairs of states with the first state from A_1 and the second from A_2. Transitions labeled with a letter in the alphabet of both automata are added to the composition if each of the automata has a corresponding transition between its states. For letters that are part of only one automaton, transitions are added to the composition if that automaton has the corresponding transition. The acceptance condition must be modified for parity automata.

2.3 Linear Temporal Logic

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 $\{\top, \neg, \wedge\}$, the alphabet of LTL includes temporal operators [30]. In this thesis, the temporal operators $\circ$ (next) and $\mathcal{U}$ (until) are letters in the formal language of LTL. Other temporal operators like $\Diamond$ (eventually), $\Box$ (always), and $\mathcal{W}$ (weak until) are also used in formulas, but they can be defined based on $\mathcal{U}$ [22]. 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.

The formal language of LTL formulas can be defined inductively. Then an LTL formula φ is generated by $\varphi = \top \mid p \mid \neg\theta \mid \theta \wedge \psi \mid \circ\theta \mid \theta\,\mathcal{U}\,\psi \mid (\theta)$, where $p \in \text{AP}$, and θ and ψ are LTL formulas. The semantics of LTL formulas can be

interpreted over words of the alphabet 2^{AP}. Whether a word $w = \sigma_0\sigma_1 \cdots \in (2^{\mathrm{AP}})^\infty$ satisfies an LTL formula φ, written $w \vDash \varphi$, is defined inductively on the structure of φ:

- $w \vDash \top$ always holds;
- $w \vDash p$ iff $p \in \sigma_0$;
- $w \vDash \neg\psi$ iff $w \vDash \psi$ does not hold;
- $w \vDash \psi \wedge \theta$ iff $w \vDash \psi$ and $w \vDash \theta$;
- $w \vDash \circ\,\psi$ iff $\sigma_1\sigma_2 \cdots \vDash \psi$;
- $w \vDash \psi\ \mathcal{U}\ \theta$ iff there is an $m \geq 0$ such that $\sigma_m\sigma_{m+1} \cdots \vDash \theta$ and for all $0 \leq i < m$ it holds that $\sigma_i\sigma_{i+1} \vDash \psi$.

The unary *next* operator $\circ$ means that the formula ψ holds in the next state. The binary operator $\mathcal{U}$ is interpreted as *until*, and means that a formula θ holds in the current or some future position of the word, and at all positions prior to that, ψ holds. The other temporal operators can now be defined as:

- $\Diamond\varphi \equiv \top\ \mathcal{U}\ \varphi$, which is interpreted as *eventually*, and means that a formula holds at some future position in the word;
- $\Box\varphi \equiv \neg\Diamond\neg\varphi$, which is interpreted as *always*, and means that a formula holds at the current position and all future positions of the word;
- $\psi\ \mathcal{W}\ \varphi \equiv \Box\psi \vee \psi\ \mathcal{U}\ \varphi$, which is interpreted as *weak until*, and means that a formula ψ holds until θ holds, or ψ holds forever.

(The propositional operators $\vee$, $\rightarrow$, and $\leftrightarrow$ are defined by $\neg$ and $\wedge$ as usual.)

The language of an LTL formula φ, denoted $\mathcal{L}^m(\varphi)$, is the set of all infinite words that satisfy the LTL formula, i.e., $\mathcal{L}^m(\varphi) = \{w \in (2^{\mathrm{AP}})^\omega \mid w \vDash \varphi\}$. The language of an LTL formula φ can be represented by a nondeterministic Büchi automaton BA, such that $\mathcal{L}^m(\varphi) = \mathcal{L}^m(BA)$ [22], [31], [32].

LTL formulas can also be interpreted over path fragments of transition systems. In that case, the LTL formula is satisfied by a path fragment if the trace of the path fragment satisfies the LTL formula. Let TS be a transition system with state set S and initial states S°, and let $\pi \in \mathrm{Frags}^\infty(TS)$. Then π satisfies an LTL formula φ, written $\pi \vDash \varphi$, if $\mathsf{trace}(\pi) \vDash \varphi$. An LTL formula

holds at state $s \in S$, written $\langle TS, s\rangle \vDash \varphi$, if all infinite path fragments starting at s satisfy φ, i.e., if $\pi \vDash \varphi$ for all $s \sqsubset \pi \in \text{Frags}^{\omega}(TS)$. The transition system TS satisfies the LTL formula φ, written $TS \vDash \varphi$, if $\langle TS, s^{\circ}\rangle \vDash \varphi$ for all initial states $s^{\circ} \in S^{\circ}$.

Two classes of interesting properties of LTL are safety and liveness, 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 means that safety properties define what shall never happen. Liveness properties cannot be disproved with finite strings of the language, thus liveness properties specify conditions that shall be fulfilled infinitely many times.

The next operator $\circ$ can be excluded from LTL to form a fragment called $\text{LTL}_{\setminus\circ}$. All stutter equivalent words satisfy the same $\text{LTL}_{\setminus\circ}$ formulas [22]. That is, let $w_1, w_2 \in (2^{\text{AP}})^{\infty}$ be two stutter equivalent words, then $w_1 \vDash \varphi$ iff $w_2 \vDash \varphi$.

2.4 Model Checking

Model Checking [22] is used to check that a formal model of a system satisfies the formalized requirements, the *specification*. The system can consist of hardware or software, but planners that interacts with their environment are considered in this thesis. The planner and its environment interacts in a feedback loop, as seen in Fig. 2.1. However, *Model Checking* (MC) does not make a difference between the planner and its environment, and treats them together as one system.

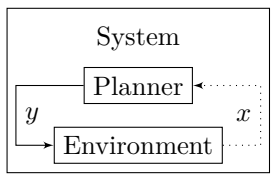

Figure 2.1: A closed-loop system with environment and planner. x is the output from the environment, which can be observed by the controller. y is the output from the planner.

The system shown in Figure 2.1 is called a *closed-loop system*; all behaviors are internal to the system. Figure 2.1 shows the types of systems that are

most relevant for this thesis. In general, the system can be composed of several parallel processes that run asynchronously and pass messages between them.

Verification of software with MC starts with the formal specification and the formal model of the system. The intended result of model checking is either a proof that the model exhibits the behavior of the formal specification, or a counterexample that demonstrates why the specification does not hold for the model. The benefit of performing model checking over reviewing and testing is that model checking is exhaustive, it is automatic, and it 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. state-space explosion problem.

In LTL Model Checking, the formal properties are expressed in LTL and the software program is modeled as a transition system, typically a Kripke structure. Let the transition system K be a Kripke structure that models a software program, and φ an LTL formula that expresses a desired behavior of K. Then a Büchi automaton B is constructed for $\neg\varphi$, and it is checked whether an accepted word of B is a trace of an allowed path of K [22]. If such a path exists, it is satisfied by $\neg\varphi$ and hence cannot be satisfied by φ. Essentially, it is checked whether the transition system K has a path specifically forbidden by φ. If so, K cannot possibly fulfill 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 more suitable for humans, and the properties in LTL. For instance, in Paper B, the language Promela is used to model the system, and the model checker Spin [33] is used to model check specifications expressed in LTL. The Promela models are automatically translated into a transition system and the specifications into Büchi automata.

2.5 Reactive Synthesis

Reactive Synthesis (RS) [24] 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 $\phi_e \rightarrow \phi_s$ [34]. A reactive module evolves in computation cycles and engages in an ongoing interaction with its environment, as illustrated in Figure 2.2. The reactive module P alternately reads inputs from the environment E and

assigns values to its outputs, possibly affecting the environment. For a comprehensive introduction to RS, refer to [35].

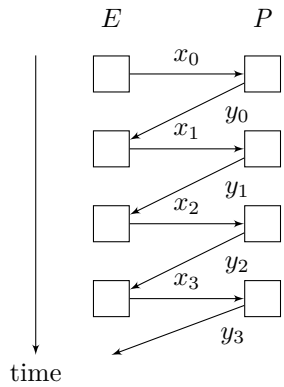

Figure 2.2: Progression of time and the turns of the environment E and the Planner P.

The planner and the environment interact in a feedback loop, as seen in Fig. 2.3. The planner observes the output x from the environment and sets its output y to affect the environment such that the specification is fulfilled.

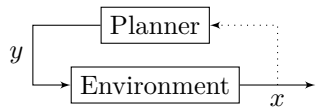

Figure 2.3: The planner is synthesized in a feedback loop with the environment.

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 synthesis can be performed as follows. The words over the alphabet $\Sigma = 2^{\mathrm{AP}_e \cup \mathrm{AP}_s}$ that satisfy the formula $\phi \equiv \phi_e \rightarrow \phi_s$ can be represented by an NBA. The NBA can be converted to a language-equivalent *parity automaton* [36], which in turn can be converted to a *parity game* [37] with two players. One player represents the environment propositions and the other player represents

the reactive module's propositions. Solving the game yields a planner that guarantees the fulfillment of ϕ [37].

The method outlined here for general LTL formulas has a worst case double exponential computational complexity [38]. However, despite the bad computational complexity, recent works have presented algorithms that are usable in practice for general LTL formulas [37]. Another successful path to tackle the computational complexity is to consider fragments of LTL.

One subset of LTL that is particularly interesting is called *General Reactivity of Rank 1* (GR(1)) [39]. This subset of formulas is useful because it is possible to perform reactive synthesis of the GR(1) fragment in polynomial time. Given an LTL formula 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 [39]:

$$\varphi \triangleq \big((\psi^e_{init} \wedge \Box\psi^e_{safe} \wedge \bigwedge_{0<i\leq J} \Box\Diamond\psi^e_{live,i}) \rightarrow (\psi^s_{init} \wedge \Box\psi^s_{safe} \wedge \bigwedge_{0<i\leq N} \Box\Diamond\psi^s_{live,i})\big), \tag{2.3}$$

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. In this thesis, Paper C and Paper E use the GR(1) synthesis tool TuLiP [40] to synthesize tactical planners.

2.6 Supervisory Control Theory

Supervisory Control Theory (SCT) [23], [41] is a model-based approach for control of *Discrete Event System* (DES). In a DES the transitions of the automata 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. It is also possible to verify that the DES satisfies the specification without synthesizing a supervisor.

In SCT, the accepting states are called *marked states*. This thesis considers SCT where the languages consist of finite strings, but there are also variants of SCT where the strings are infinite [42]. The requirements on a system can be formalized as marked states in the plant, or as separate specification automata with their own marked states.

In the closed-loop interaction of the supervisor and the plant, depicted in Figure 2.4, the plant generates all events under control of the supervisor that can disable some of them. Thus, the alphabet Σ is split into two sets; the *controllable* events $\Sigma_c \subseteq \Sigma$ that can be disabled by the supervisor, and the *uncontrollable* events $\Sigma_{uc} = \Sigma \setminus \Sigma_c$ that cannot be disabled.

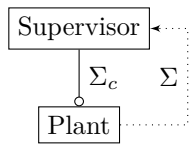

Figure 2.4: The supervisor observes events generated by the plant, and may control the plant by disabling controllable events.

A supervisor P for a plant G must never disable any uncontrollable event of G, it must be *controllable*. This means that if G allows an uncontrollable event after a word that is in the language of both P and G, then P must allow that uncontrollable event. Formally, $\mathcal{L}(P)\Sigma_{uc} \cap \mathcal{L}(G) \subseteq \mathcal{L}(P)$ must hold [23].

One property of the closed-loop system that is of interest in SCT is whether the resulting DES is *blocking*. A DES is blocking if there exists some reachable unmarked state from which no finite path fragment reach a marked state. In terms of languages, for a supervisor P and a plant G, the composed system $P \parallel G$ is blocking if $\mathsf{pfx}(\mathcal{L}^m(P \parallel G)) \subset \mathcal{L}(P \parallel G)$, and it is *non-blocking* if $\mathsf{pfx}(\mathcal{L}^m(P \parallel G)) = \mathcal{L}(P \parallel G)$. By definition, $\mathsf{pfx}(\mathcal{L}^m(P \parallel G)) \subseteq \mathcal{L}(P \parallel G)$, so for evaluating non-blockingness it suffices to show $\mathcal{L}(P \| G) \subseteq \mathsf{pfx}(\mathcal{L}^m(P \| G))$ [23].

To ensure that the closed-loop system satisfies the specification, any word possible in the closed-loop system must be allowed by the specification. 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. For a supervisor P, plant G, and a specification K, this means that it must be ensured that $\mathcal{L}(P \| G) \subseteq \mathcal{L}(K \| G)$ and $\mathcal{L}^m(P \| G) \subseteq \mathcal{L}^m(K \| G)$.

Basically, supervisor synthesis is an iterative removal of states and/or transitions of an initially calculated supervisor "candidate". Practically, this candidate is calculated from $G \| K$ [23]. 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 controllable, non-blocking, and *maximally permissive* supervisor. Maximal permissive means that the supervisor disables as few events as possible.

With the introduction of *updates* to NFAs it is possible to restrict the plant's behaviors by adding updates to the plant. In that case the synthesis algorithm does not have to generate a supervisor as a separate automaton, but can add to the plant new updates that control the generation of controllable events.

Binary Decision Diagram (BDD) [43] can be used in the synthesis algorithm for the purpose of adding updates to plants. 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. The approach presented in [44] uses partitioning techniques to further stretch the ability of the synthesis procedure. In Paper C, Paper D, and Paper E, the SCT tool Supremica [45] is used to synthesize supervisors.

2.7 Note on Terminology

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 [40] and Supremica [45] are compared, so this section gives a short note on the terminology used in these fields.

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; 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 sensors or actuators, as seen in Fig. 1.1. 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 seen in Figure 2.1. The formalization of the requirements is called the *specification*, and this thesis will follow that convention.

The terminology of RS differs from MC. The goal in RS is to *synthesize* the planner, or *reactive module*, such that it *guarantees* the fulfillment of the requirements as long as the *environment* model fulfills the *assumptions* put on it [40]. The assumptions are specifications on how the *environment* produces the input x, as shown in Fig. 2.3. If the environment assumptions are not fulfilled, then the planner does not need to fulfill the specification.

Since RS aims to synthesize the planner, there is never a model of the planner in RS, but a planner *specification* derived from the requirements. In RS terminology there is also no environment model, but an environment *specification* that states the assumed behaviors of the environment. 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 *plant* and *specifications*. In general, the plant models the environment, and synthesis automatically computes a supervisor according to the given specification. For verification, the plant models the environment and the supervisor in a closed-loop, and it is determined whether this feedback system fulfills the specification. The specifications are the formalizations of the requirements. However, the plant can also have markings on states, which is a formalization of a certain type of requirements that express the possibility to reach those states. The goal of synthesis with SCT is to construct a *supervisor*.

There is also different terminology in use for languages and transition systems. The alphabet Σ of automata generally consists of letters, but in the specific case of SCT the elements of the alphabet are called *events*. For tran-

sition systems the alphabet consists of *actions* [22] or, like in Paper F, they are *control inputs*. Furthermore, the states in the acceptance condition F are commonly referred to as *accepting* states, whereas SCT calls them *marked* states. Likewise, the *accepted* and *marked* language of an automaton refer to the same concept.

Functions that choose the next state, output, or action based on a transition system's history are referred to as *controllers*, *strategies*, *policies*, or *reactive modules*. They are all defined as a function observing a sequence of states or letters, and outputs a decision on where the system shall transit next. *Supervisors* are similar, instead of outputting a decision, they disable events.

To the extent possible, this thesis will refer to the synthesized entity as a planner, its surroundings as the environment, and the formalized requirements as the specification.

CHAPTER 3

How safe?

In order to deploy Automated Vehicles on public roads, the automotive industry must present compelling arguments that the vehicles are sufficiently safe [15]. Here, and henceforth, "sufficiently safe" means that a vehicle or system does not cause an unacceptable high risk of harm. The safety arguments are often compiled into a *safety case* which is "a structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment [46]." There are three principal elements in a safety case: requirements, arguments, and evidence [47]. The safety case details, among other things, the processes used to govern the safety work, the identified hazards and risks, the break-down and implementation of the requirements that mitigate the identified risks, the verification and validation plan and its results, and an argument justifying that all of these activities together give evidence that the system is safe [21], [48].

Systems that aid the driver in their driving task through warnings or by performing the entire or parts of the driving task, are generally referred to as *driving automation systems*. The amount of work that goes into the safety case is largely governed by the degree of driver involvement in the driving tasks

and the extent of the situations that the driving automation system handles. Based on these capabilities, driving automation systems are popularly divided into six standardized levels of automation, as detailed by SAE J3016_202104 *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles* [2]. Though this standard is not about safety, it defines the terminology that is used in some safety standards.

Two widely accepted automotive safety standards that recommend processes for how companies can manage and systematically work with safety are ISO 21448 *Road vehicles – Safety of the intended functionality* (SOTIF) [48] and ISO 26262 *Road vehicles – Functional safety* (FuSa) [21], where the former addresses risks of performance deficiencies in system features, and the latter addresses risks caused by malfunctions. Both standards focus on limiting risk of harm that originates from deficiencies and failures of E/E systems, such as sensor limitations, algorithmic faults, and hardware faults. They explicitly exclude risk of harm that stems from, for instance, fires caused by malfunctioning hardware, unauthorized access due to lacking cyber security, etc. This delimitation applies also to this thesis.

The standards help with structuring the safety work and they provide recommendations for which methods to apply in different stages of the development process in order to limit the risk of deficiencies or failures that have harmful consequences. However, in the end it must be shown that the residual risk is sufficiently low [48], i.e., the risk of harm that stems from unknown and unanalyzed situations. What "sufficiently low" means is of course subjective, but it will likely be affected by regulations and the current traffic accident statistics [49]. For an automated driving system that completely replaces a human driver, i.e., a fully automated vehicle, a tremendous effort is required to validate such a target on a complete-vehicle level. For instance, as noted by [13], if a fleet of 100 automated 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 [15].

3.1 Levels of Driving Automation

The six levels of driving automation categorize driving automation systems based on which agent performs the *dynamic driving task* (DDT) and the DDT fallback, and how extensive the *operational design domain* (ODD) is [2]. The DDT is the normal driving task and it consists of two parts, the lateral and longitudinal control, as well as the *object and event detection and response* (OEDR). Lateral control typically refers to operation of steering, whereas longitudinal control refers to operation of engine torque and brake torque, i.e., the normal motion control of the vehicle. The OEDR task consists of monitoring the driving environment to detect objects and events that affect the DDT, and properly respond to such objects and events, at all times. Detecting a preceding vehicle and determining a proper speed and distance to it, or identifying a flat tire and safely stopping at the side of the road are two examples of OEDR tasks. With these definitions of OEDR, and lateral and longitudinal control it is clear that the DDT consists of the tasks of tactical and operational planning, but not strategic planning.

The ODD is the operational conditions under which a driving automation system is designed to perform the DDT. The conditions that define the ODD can be for instance road type, lighting conditions, level of precipitation, traffic conditions, etc. Driving automation systems can be activated to perform the DDT once the vehicle has entered their ODD, and they stop performing the DDT when the vehicle exits their ODD. Some operational conditions that make up the ODD can be transient and intermittent, and in that case the driving automation system does not have to deactivate if reliable operation of the DDT can still be performed; a system that requires high quality lane markers for sustained operation might still operate reliably if the quality of the lane markers are low for a short period of time.

On exiting the ODD there are two options for the driving automation system: either the system itself performs a DDT fallback, or it cedes operation to the driver who then performs a DDT fallback[1]. In addition, a DDT fallback is performed if the vehicle suffers from a failure that prevents the driving automation system from reliably performing the DDT. As with the ODD exit, the driving automation system either performs the DDT fallback itself, or cedes operation to the driver who performs the DDT fallback. One allowed

[1]The standard allows a remote operator to perform the DDT fallback as well, but that distinction is not important in the context of this thesis.

DDT fallback is to perform a *minimal risk maneuver* (MRM) to reach a *minimal risk condition* (MRC), where the MRC is a stopped position where the risk of harm is acceptably low. The topic of Paper B is to use formal verification to prove the correct operation of a tactical planner that is responsible for activating the DDT fallback.

The six levels of driving automation are defined based on which agent performs the different parts of the DDT, which agent is ready to perform the DDT fallback, and the extent of the ODD. The higher the level, the less responsibility is put on the driver. The levels distinguish between a limited and an unlimited ODD. A limited ODD excludes some driving situations, while an unlimited ODD contains all possible driving situations.

Level 0 *No Driving Automation:* The human driver performs the entire DDT and is responsible for the OEDR.

Level 1 *Driver Assistance:* A driving automation system performs sustained lateral **or** longitudinal control in a limited ODD. The human driver is responsible for the other tasks and is ready to perform a DDT fallback.

Level 2 *Partial Driving Automation:* A driving automation system performs sustained lateral **and** longitudinal control in a limited ODD. The **human driver** is responsible for **OEDR** and is ready to perform a DDT fallback.

Level 3 *Conditional Driving Automation:* A driving automation system performs the **entire DDT** in a limited ODD, but a **human driver** is ready to perform a **DDT fallback**. The system may perform a DDT fallback itself, but might not do so in every situation.

Level 4 *High Driving Automation:* A driving automation system performs the entire DDT in a limited ODD, and **must perform a DDT fallback** when needed. It cannot be expected that a human driver is available to perform a DDT fallback, but it is allowed to ask a driver to perform the DDT fallback, for instance, at the ODD exit. However, if a driver fails to assume operation of the vehicle, then the system must perform the DDT fallback.

Level 5 *Full Driving Automation:* A driving automation system performs

the entire DDT in an **unlimited ODD**, and must perform a DDT fallback when needed.

Essentially, the levels categorize driving automation systems based on driver involvement and driver responsibility. It is only for Level 5 that the ODD has a determining role. However, for the levels below Level 5, the extent of the ODD also provides a scale on which systems fall, so it can be seen as the levels are one axis and the ODD is another axis on which driving automation systems can be categorized.

Systems at levels 1 and 2 are referred to as *driver support* systems, whereas systems in levels 3 to 5 are referred to as *Automated Driving Systems* (ADS). The term Automated Vehicle roughly refers to vehicles equipped with an ADS, and Fully Automated Vehicle refers to Level 5 systems. This thesis is mainly focused on ADS, where Paper B and Paper E specifically consider a vehicle with an ADS in Level 4. However, many of the methods and conclusions of this thesis are probably applicable also to driver support systems, and some systems in Level 0.

3.2 Driving Around the World

Several companies working with driving automation systems publish reports that detail their efforts to ascertain safety [50]–[54]. They also publish how long they have driven with their field test vehicles (or customer vehicles in the case of Tesla). Waymo had in 2017 driven 10 million miles [55], and increased that to more than 20 million miles in 2021 [54]. They had additionally driven more than 15 billion miles in simulated environments. Tesla drivers had driven 1 billion miles in 2018 [56], and increased that figure to 3 billion miles in 2020 [57]. Yandex drove 1 million miles until 2019 [58], and increased it to in total more than 6 million miles in 2021 [59]. Uber had driven at least 2 million miles in 2018 [60], [61]. All of these companies, except Tesla, are field testing vehicles with ADS in SAE J3016_202104 Level 4. Tesla's system is available for consumers and relies on the driver performing parts of the OEDR, so it is a Level 2 driver support system.

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 vehicle is at least as safe as a human driver (in the U.S.) it needs to drive 255 million miles with no fatalities [15]. As can be seen above, most ADS companies fall short

on this target with at least one order of magnitude. Tesla seems to fare well since they have tested on 12 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, for instance [15]. And there have been fatalities where, for instance, Tesla and Uber cars have been involved [62]–[64].

It is also possible to consider the failure and disengagement rate of the ADS that are tested. During testing, there is a safety driver in the vehicle who is ready to disengage the ADS and assume operation of the vehicle if there is a system failure or an unsafe situation. The rate at which these disengagements occur can illustrate the difficulty of validation. For instance, in 2020 in California, Waymo's ADS had 0.0333 system failures or safety-driver disengagements per 1000 miles [65]. This means a failure or disengagement every 30 thousand miles, and consequently 8500 failures or disengagements per 255 million miles, none of which are allowed to cause a fatality if the safety driver is not to intervene.

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

3.3 ISO 21448 and ISO 26262

As mentioned above, ISO 21448 (SOTIF) and ISO 26262 (FuSa) are two safety standards that apply to the automotive industry, where SOTIF addresses the safety of the intended functionality and FuSa addresses functional safety, i.e., reducing risk of malfunctions. FuSa applies to any automotive E/E system, whereas SOTIF's aims is to complement FuSa and provide safety processes for ADS, driver support systems, and other E/E systems that sense the external environment (and thus often support part of the OEDR somehow). Both standards structure the safety work by breaking it down into different work packages on different subsystems and for different system components, and thereby manage the complexity of the safety case.

The two standards begin the work in similar ways by identifying possible hazards. These are behaviors that may cause harm. Each hazard is considered in all the operational situations of the system, its ODD, and the combination of the hazard and an operational situation is called a hazardous event. The

criticality of each hazardous event is then based on how severe the consequences of that event is (severity), how often the vehicle is in the operational situation of the hazardous event (exposure), and how likely it is for an agent other than the safety-critical functionality to detect and respond to the hazardous event (controllability). For instance, one identified possible hazard could be the omission of braking, and this hazard together with the operational situation of approaching a stationary vehicle, resulting in a collision, is a hazardous event. 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 to avoid the collision, which, among other things, is influenced by the automation level.

The purpose of the ISO 26262 standard is to prevent the hazardous events caused by malfunctions, 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 all safety goals that are attached to the hazardous event, and the ASIL will guide what safety efforts that are needed during development to ascertain the fulfillment of each safety goal. The ASIL of a safety goal can be interpreted as the criticality of the safety goal, and a higher level means more critical and that a lower frequency of violation is accepted.

When all safety goals have been formulated, the standard follows a V-model [21]: an architecture with subsystems is developed for the system, and the safety goals are broken down into functional safety requirements, allocated to the subsystems.

For driving automation systems in Level 1 to Level 5, the exposure might be heavily influenced by the system design and performance. Such systems perform (parts of) the DDT, and thus may affect the exposure to hazardous events through their actions [66]; keeping a close distance to the preceding vehicle on a highway when an on-ramp joins, might, for instance, increase the exposure to close cut-ins from other vehicles. In SOTIF, the solution is to set a target exposure rate of the triggering events. If that target is judged too difficult to attain, the system must be redesigned and potential hazards must be identified for the new design. The redesign can typically

be performed according to the V-model as with FuSa, with a break-down of new requirements to subsystems. Otherwise, if the target is judged to be acceptable, then the fulfillment of the target must be shown by verification and validation. The verification and validation tasks and their corresponding target exposures can also be broken down to the subsystems.

It has been suggested that target exposure rates should be set similarly to SOTIF for ADS in FuSa. The safety goals would then be based on *consequence classes* and *incident types* instead of hazardous events to form a *Quantitative Risk Norm* (QRN) [66]. In QRN there are different consequence classes for different ranges of severity, and each consequence class has a target frequency of occurrence. Incidents are undesired events, including accidents, and incident types are different categories of incidents. An incident type contributes to one or several consequence classes, and one consequence class can have several contributing incident types. The combination of one incident type and one consequence class gives rise to a safety goal with a target maximum frequency. After the safety goals have been defined in this manner, the usual FuSa process follows [66]. The example Safety Case in Paper A follows this approach.

From a safety point of view, ADS are increasingly difficult to develop because of three mechanisms. Firstly, as the ODD is expanded, so is the complexity of the technical solution. Even if the highest ASIL or lowest target exposure of all the safety-critical requirements on a specific subsystem is the same, the increased complexity means more safety effort. Secondly, expansion of the ODD means more hazardous events with higher severity and exposure. These increases lead to higher ASIL, or lower target exposure, and thus again greater safety effort. Also, identifying a high proportion of all the triggering events becomes increasingly hard since the number of potential system deficiencies increases as the ADS must handle more driving situations. Thirdly, the goal is to let the system perform the OEDR, or parts of it, and this increased reliance on the system causes lower controllability [67], which also means higher ASIL or lower target exposure.

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 [68], [69], and it is difficult to implement fault free code [70]. 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 requires, 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 [22].

3.4 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. SOTIF and FuSa provide processes that can support that aim by judging risks in different situations, and recommend how to mitigate them to acceptable levels. Part of the process is to break down the safety-critical requirements to individual components. The break down 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 are 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 on the software level. However, the standard does not instruct how formal methods can be applied on a functional level for feedback systems, 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.

3.5 Formal Methods in Safety Argumentation

There are several studies that contribute with approaches on how to structure an argument based on formal methods in the safety case [71]–[74]. However, these studies concern formal methods applied to implemented software, and not to the functional characteristics of feedback systems. Thus, one of the challenges to solve when using formal methods to ensure safety of tactical planners is how to structure an argument in the safety case such that the formal proof may be used as evidence.

One aspect of the safety in this context is that the tool that provides the formal proof must also be subject to a convincing argument of its correctness. A systematic approach to structure such an argument is suggested by Habli and Kelly [75]. For program verification, such an argument must include evidence that the tool's model of the semantics of the programming language is correct with respect to the behavior of the compiled code. This evidence is related to the tool and can in many cases be gathered once.

For tactical planners, however, the formal model typically is not so detailed as to capture the semantics of software, but rather is coarser to efficiently reason about more abstract behaviors over longer time horizons. In general, this makes the formal model dependent on the specific problem, so the evidence of correct models must be handled per instance, and cannot be gathered only once. Other aspects of the correctness of a tool, such as correctness of the proof system and the implementation of the tool, can still be gathered once.

One approach for structuring the safety argumentation for ADS based on formal methods is presented in Paper A. The paper uses the *Goal Structuring Notation* (GSN) [76] to illustrate the structure of the argument. GSN is a set of graphical elements and relations that can be used to graphically show how an argument is structured in a *GSN goal structure*, which is an acyclic directed graph. The graphical elements of GSN are shown in Fig. 3.1. Rectangles represent *goals*, and these are the elements that correspond to requirements whose fulfillment must be supported by compelling evidence. Parallelograms represent *strategies*, which are arguments that justify how a goal is supported by other evidence. Circles represent *solutions*, which are concrete pieces of evidence. For instance, a formal proof may be taken as evidence for a solution. Ellipses labelled with capital "A" and rectangles with rounded corners represent *assumptions* and *contexts*, respectively. They both indicate the setting of the element that they are related to, and they implicitly apply to

elements that provide evidence for the elements that they are directly related to. GSN includes the two relations *SupportedBy* and *InContextOf*, which are represented by arrows with solid and hollow heads, respectively. The diamond is used to decorate elements that are yet to be developed into a structure with supporting evidence.

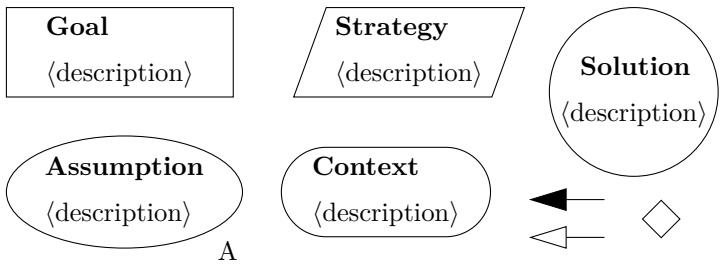

Figure 3.1: The core elements of a GSN goal structure. The solid and hollow arrowheads denote the *SupportedBy* and the *InContextOf* relationship, respectively. The diamond indicates an undeveloped element.

A GSN goal structure for a formal model and specification can be seen in Fig. 3.2. The goal is for the formal model to fulfill the specification in the form of a formal guarantee that expresses that crashes must not occur, and this is captured in G1. Fulfillment of G1 is supported by the formal proof, which is referred to by a unique ID. However, the context of the goal G1 ought to be given explicitly, i.e., the circumstances under which the goal is fulfilled. The context of the goal includes three formal models that describe the behaviors of the subsystems shown in Fig. 1.1. Without the context of the formal models to which the proof is referring, the proof has no meaning. Clearly, the formal guarantee that is proven to be fulfilled is also an important part of the context. Lastly, an ADS operates in an ODD that describes the intended operating environment. A formal model of the ODD describes how the ADS interacts with its environment, and in Fig. 3.2 it is assumed that the formal model of the ODD includes the environments in which the ADS will operate.

Paper A argues that the correct realization of the subsystems implies the correctness of the ADS, and at its core, the approach in Paper A splits the argument in two parts. One part is concerned with the evidence of the correct realization of the subsystems in Fig. 1.1, and the other part is an argument

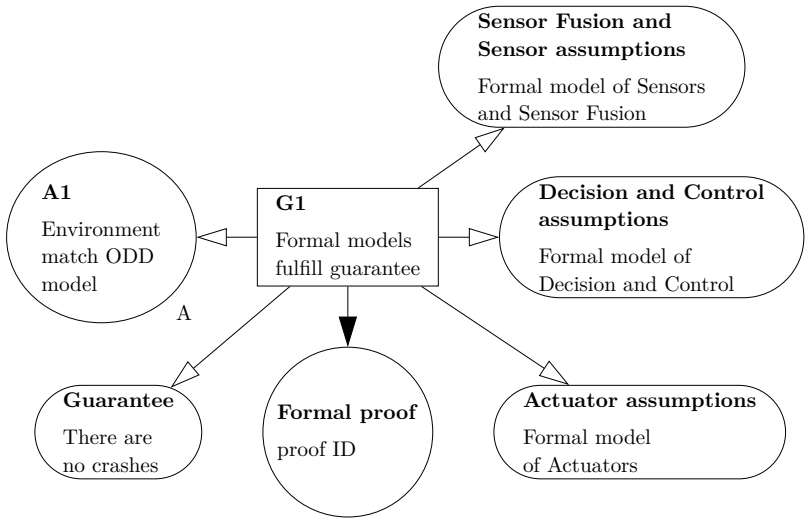

Figure 3.2: GSN goal structure illustrating how a formal model and specification can be argued to support the goal of fulfilling the specification.

based on the structure in Fig. 3.2. The argument is illustrated in Fig. 3.3, where evidence supporting that the ADS cannot crash is split into two separate arguments, the strategies S1 and S2. S1 represents the evidence that the subsystems are implemented correctly. As can be seen in the figure, correctness of the subsystems means that there is evidence for the fulfillment of the goals G3, G4, and G5, where each goal states that the subsystem's behaviors do not violate the behaviors of the corresponding formal model. A GSN goal structure showing the entire break-down from a safety goal onto subsystems is shown in Fig. 3 in Paper A.

Assume that those three goals are fulfilled. If it can be proven that the formal models fulfill the guarantee, then G1 must be fulfilled in the ODD, because the guarantee is a formalization of G1. Thus, evidence that G2 is fulfilled is also evidence that G1 is correctly broken down into G3, G4, and G5, and this is the argument being made by the strategy S2.

The three goals of the subsystems are undeveloped and must themselves be supported by a compelling safety argumentation. The ISO 21448 and ISO 26262 standards provide means and processes for structuring such argu-

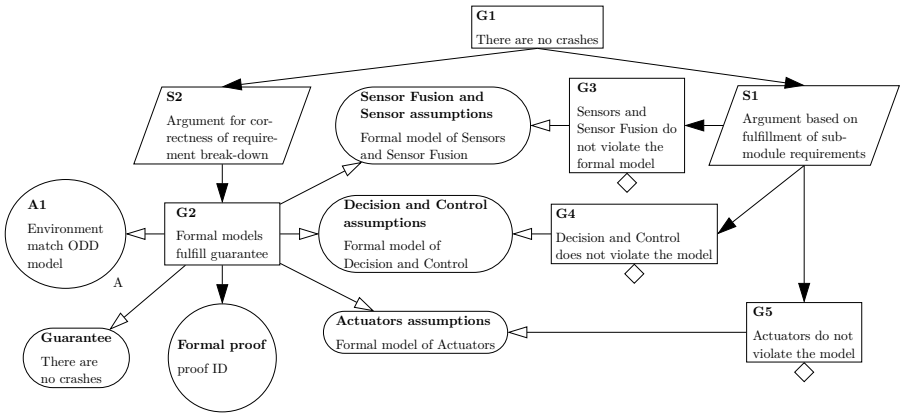

Figure 3.3: GSN illustrating an argument for the correct break-down of the goal G1 onto the goals G4, G5, and G6. See Fig. 3 in Paper A for more details.

ments.

The contributions of Paper A suggest how the results of formal methods applied to tactical planners fits in a compelling safety argument.

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 [25], [77]–[81]. Tactical planners are special because of three characteristics;

- they make discrete decisions in environments where they interact with other road users;
- they make the decisions while moving, so the vehicle's position changes continuously; 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 require special consideration because temporal specifications are needed to describe the desired interactions, and the level of abstraction of the states have to be chosen carefully.

This chapter describes how these 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. The results and conclusions in this chapter are mainly based on Paper B, Paper C, Paper D, and Paper E, but some parts are also from Paper A.

4.1 Safety of Minimal Risk Maneuver

Recall from Chapter 3 that an ADS performs the Dynamic Driving Task (DDT) in a specified Operational Design Domain (ODD). For Level 4 ADS it is also required to perform the DDT fallback, also called a Minimal Risk Maneuver (MRM), to reach a Minimal Risk Condition (MRC). The MRC is typically a state where the vehicle is stationary and preferably some distance away from the active part of the road so that the risk of encounters with other traffic is low. As a consequence, the MRM usually includes braking in a controlled manner and maneuvering out of the active traffic lanes. The need to activate an MRM is prompted by events that compromise the safety of sustained operation of the DDT. Leaving the ODD is one such event, but events such as internal failures of the ADS may also trigger the activation of an MRM.

Paper B investigates how formal verification can be used to ensure correctness of the activation of an MRM for an AV. The task of the vehicle is to complete a transport mission starting from a parking lot and ending in another parking lot. Between the parking lots is a road network that the vehicle has to navigate through. The safety aspects of the MRM activation is handled by a tactical planner, which unfortunately is denoted as a safety supervisor in Paper B. The safety planner coordinates nominal planners and controllers, and the safety planner must activate the *safe-stop trajectory planner* in the event of a failure of any of the nominal planners. The safe-stop trajectory planner continuously monitors the road and evaluates several different trajectories to reach positions near the road where reaching an MRC might be suitable. The trajectories that reach suitable positions for an MRC are updated regularly to be available for the safety supervisor to activate when needed.

In the particular study of Paper B, the considered failure events were loss of localization capability caused by a GPS sensor failure, and failures of the nominal planners to plan new paths when the vehicle is close to the end of the current path. Loss of localization capability prompts an MRM because the ADS cannot know whether the current location is part of the ODD, and because the paths planned by the nominal planners are not safe when the

current position of the vehicle is not fully known. The failure of one of the planners to plan a new path prompts the activation of an MRM because driving without a path is unsafe, and stopping outside of the active lane is preferable to stopping in the middle of the road.

Formal verification is a versatile tool for proving absence of faults, which is experienced in Paper B. Paper B uses Model Checking simultaneously with model-based design in an iterative process to develop the safety supervisor. The system is modeled in Promela and verified with Spin [33]. As discussed in Section 2.7, the model of the supervisor and its environment are merged and there is no precise distinction between the two. In this context, since the correctness of the supervisor is the main focus, all other subsystems of the ADS and the environments defined by the ODD are considered the environment of the model of the supervisor. The different subsystems are two different nominal path planners, one safe-stop trajectory planner, one nominal trajectory planner, the localization module with the GPS sensor, a low level controller, and the vehicle dynamics, all of which can be seen in the architecture in Fig. 1 on page B6 in Paper B. The model of the implemented supervisor is proven to be correct assuming that the models of the other subsystems and the ODD fulfill the properties of the real environments that the ADS is deployed into.

The formal specification of desired properties is given entirely in LTL. As noted in Paper B, the implementation of the supervisor was easy to integrate with the rest of the vehicle's systems, and no errors were found during simulation or in-vehicle experiments. Although the simulations and in-vehicle experiments were not exhaustive, the smooth integration gives some indication that the process with alternating design, implementation, and model checking contributes to less errors than if it had not been used.

In terms of safety argumentation, Paper B does not explicitly detail how the results and the artifacts contribute to a safety case. However, many of the important artifacts and pieces of evidence at the behavioral level of the ADS as detailed in Paper A are produced. The formal proofs of the formalized requirements show that the models of the subsystems together fulfill the requirements. Implicitly, the formal models are then taken to be the requirements on the subsystems, although the approach is both top-down and bottom-up. As realizations of several subsystems already were available at the start of the study in Paper B, the formal models were initially based on the implementations of those subsystems, but during the development of the su-

pervisor there were changes made in the formal model that prompted changes in the realizations.

To show that the realizations of the subsystems fulfilled the behaviors of the formal models, the subsystems of the ADS were tested separately by unit tests, together in simulations, and together in in-vehicle tests. The tight connection between the supervisor model and its implementation also meant that the proofs of the correctness of the formal model can help justify the correctness of the implementation. Still, as noted in Paper B, the need to synchronize the formal model and the implementation of the supervisor is a weak point in the process, which requires other means of gathering evidence of its correctness. In Paper B this is done by testing in simulation and in-vehicle tests.

The study in Paper B shows that it can certainly be useful to use Model Checking for LTL formulas in the development of safety-critical tactical planners for AVs. 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 [22], 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.* [26], is when a specification does not hold and the formal verification reports a counterexample that is difficult to remedy manually.

Furthermore, in Paper B there are several different requirements that must be fulfilled by the models. Theoretically, these requirements can all be conjuncted and model-checked together. Practically, that approach is not feasible in general because the size of the Büchi automaton used in Model Checking is worst case exponential in the size of the LTL formula [31].

Not conjuncting the formulas has other beneficial effects because different LTL formulas refer to different atomic propositions, and they use different temporal connectives. This can be exploited by the Model Checker to effectively reduce the size of the problem instance. For instance, as stutter trace equivalent path fragments satisfy the same $\mathrm{LTL}_{\setminus \circ}$ formulas, the Model Checker can potentially collapse many states of the formal model with the same atomic propositions, thus likely reducing the problem size. Conjuncting several LTL formulas may then reduce these opportunities.

Model Checking each requirement separately is a drawback when the requirements are not fulfilled by the model. A failed verification results in a

counterexample that may be used to find and help correct faults in the model. However, a correction for one requirement might also invalidate another requirement. These drawbacks mean that manual effort is required to iteratively design correct models, as was done in Paper B. Formal synthesis automates this process to find a model that satisfies all the requirements.

4.2 Comparison of Reactive Synthesis and Supervisory Control Theory

The purpose of this thesis is to investigate whether and how formal methods, and in particular formal synthesis, can be used to design safe tactical planners. Part of the investigation into formal synthesis is performed in Paper C, in which two small case studies are set up to compare Reactive Synthesis (RS)[24] and Supervisory Control Theory (SCT)[23], [41] from a more general modeling perspective. Though Paper C considers the two specific formalisms GR(1) [39] and SCT and the two specific tools TuLiP [40] and Supremica [45], the interesting result for this thesis is the comparison of some of the different characteristics of the formalisms. The comparison in Paper C is based on two case studies; a turn-based game and an automated vehicle scenario.

From a technical and theoretical standpoint, RS and SCT differ quite significantly, but studies have shown that most aspects can be translated from one formalism to the other [82]–[84]. However, maximal permissiveness cannot in general be enforced in RS, and liveness properties cannot in general be enforced in SCT, at least not for the formulations considered in this thesis. In contrast to the theoretical studies, Paper C studies what these differences mean for a practitioner when modeling. The two problem instances studied in Paper C were chosen to focus on the modeling differences of the formalisms in general, and not specifically focused on automotive applications.

One fundamental difference between TuLiP and Supremica is the modeling format. TuLiP has support for synthesis with LTL specifications, albeit the restricted fragment of GR(1), while 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 Paper C,

Paper D, and Paper E. Based on the theory of formal languages in Chapter 2 and the results in Paper C, it seems like the modeling formats are sufficiently similar to allow expressions of similar structure in an automata sense.

4.3 Patching with Supervisory Control Theory

Paper C compares RS and SCT from a modeling perspective and does not focus on the applicability of formal synthesis for tactical planners. This applicability is investigated in Paper D and Paper E. The most relevant results of Paper D are presented in this section, and the most relevant results of Paper E are presented in the next section.

The goal of Paper D 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 D). The manually implemented code was modeled by Zita *et al.* [26] and formally verified with Supremica [45]. The result of the verification exposed several faults, of which one was very elusive in the attempts to manually correct it. Paper D 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 obtained from Supremica in the scope of Paper D 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 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. Typically, a planner cannot restrict its inputs, so this synthesis result it highly undesirable.

[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 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.

4.4 Reactive Synthesis and Supervisory Control Theory for Realization

Paper D and Paper E are similar to each other, but where Paper D applies synthesis with SCT on an existing model, Paper E investigates how the formal models and requirements from Paper B can be used as a basis for synthesis with RS and SCT. The purpose of Paper E is to find benefits and drawbacks of using RS and SCT as modeling languages, both compared to each other, but also compared to the Model Checking performed in Paper B. The comparison is made on models in the tools TuLiP and Supremica. The intention is not to compare the performance of the tools themselves.

The hypothesis of Paper E is that the synthesis effort would be low because requirements and models are already available from Paper B. This is true for the requirements.

The requirements are directly usable for RS, except that some LTL formulas used in Paper B cannot directly be formalized in the GR(1) fragment. For instance, there is a requirement specifying that the vehicle must reach the goal infinitely often or it must eventually be stopped safely forever. This is formalized as $\square\lozenge g \vee \lozenge\square s$, where g represents the goal position and s represents being safely stopped. $\lozenge\square s$ cannot be expressed in GR(1). This was dealt with by conjuncting the specifications $\square\lozenge s$ and $\square(s \rightarrow \circ s)$. The former means that s must hold infinitely often, and the latter means that once s holds, then s holds forever in the future. The combination of these specifications satisfy $\lozenge\square s$. However, this is a stricter specification than $\lozenge\square s$, because $\lozenge\square s$ allows behaviors of the kind $(s)(\neg s)(s)\cdots(\neg s)(s)(s)\cdots$, while $\square\lozenge s \wedge \square(s \rightarrow \circ s)$ does not. This is not a problem in Paper E, however, because the intention of the specification is that once s holds, it shall hold forever in the future.

The specifications for SCT were implemented as automata with the LTL specifications as blueprints. The requirements have simple formalizations in Supremica, but because the semantics of LTL and automata in SCT differ, the specifications in Supremica does not express precisely the same behavior as in LTL.

The models, on the other hand, is not as easily used for synthesis. The

models in Paper B have different level of abstractions. The model of the vehicle describes the relation between position and speed, whereas the tactical planner makes decisions on a much higher level. In Paper E it is found out that this mix of abstraction levels is difficult to handle in synthesis with TuLiP and Supremica. The tactical planner automatically gets access to the low level states of the vehicle model, which means that the planner is synthesized to operate in environments with the exact same length of the planned paths. The planner must be more abstract than that to be sufficiently general to be used in more diverse environments. It is also computationally demanding to handle all the positions in the path in the synthesis algorithms. A higher level of abstraction for the vehicle model is chosen in Paper E, but this puts more demands on ensuring that the model and the low-level controller are correct.

In the following sections, combined results of Paper B, Paper C, Paper D, and Paper E are presented. That is, conclusions and results that bridges across all of these four papers.

4.5 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 formal specifications that all have very precise and formal syntax and semantics, but with several interacting formal specifications 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) [85], [86]. 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 result 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 vehicles are safe in certain operational domains. As described in Section 4.6, 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 D where the results would otherwise not have been possible to understand. Paper E 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.

Paper E 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 D, which only concerns SCT (more specifically Supremica), Paper E also brings these conclusions into the field of RS, or at least synthesis with the GR(1) fragment, by evaluating synthesis in TuLiP [40] side-by-side with Supremica [45]. 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 E 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.

Similarly, Paper C uses visual inspection of the supervisor and the reactive module to ascertain that they represent the same solution in its first case study. As the state spaces are relatively small, visual inspection is possible.

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 heavily reduced. The comparisons in Paper C that require evaluation of the results are instead performed with simulations. Simulations are enough to obtain 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 collapsing (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.

4.6 Verification vs. Synthesis

The main conclusion of both Paper D and Paper E 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.* [26] provide plants and specifications in Supremica to find a fault in the LSM. The approach of Paper D 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. However, what matters for synthesis 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 E 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 in Section 4.5 above, explaining when the synthesized planner works is an important part of a credible safety argument [15], and separate formal environment assumptions can simplify that argumentation.

The specifications expressed in LTL in Paper B can be used with almost no changes in the synthesis of the reactive module in Paper E. Minor changes are needed to account for the limitations that GR(1) brings. The translation of the LTL formulas 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 specifications are needed to synthesize a tactical planner with the same behavior as the manually implemented one.

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 matter simplifies the process.

4.7 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 use. As already indicated in Chapter 2, 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 mainly covers the differences in modeling for synthesis.

The GR(1) fragment, which the TuLiP model is restricted to, limits which

formulas 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) formulas and extra atomic propositions in AP. Piterman *et al.* [39] demonstrate how 'until' can be specified in GR(1) synthesis, and Paper E has examples of how other non-GR(1) formulas are expressed. For instance, the property of $\square(p \rightarrow \square q)$, where $p, q \in AP$, can be expressed as:

$$\square(q \rightarrow \circ q) \tag{4.4}$$

$$\square(p \rightarrow q)\ . \tag{4.5}$$

The formula (4.4) expresses that if q holds at one step, then it also holds in the next. This ensures that once q holds, it will hold forever in the future. In (4.5) it is expressed that q must hold whenever p holds. Combined with (4.4), this ensures that q holds forever in the future whenever p holds.

Hence, as indicated by both Paper C and Paper E, the restrictions of GR(1) do 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 2.6 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 prevent other uses, as illustrated in [87].

The simple stick-picking game modeled in Paper C, shows 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.

The tactical planners synthesized by Supremica in Paper C and Paper E show the effects that maximal permissiveness has. In many states of these supervisors several controllable events are allowed to fire. It is up to the plant in its role as *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 the enabled ones all keep the system in a safe set of states if fired. This can be 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.

Reactive modules are not maximally permissive. The reactive modules that are synthesized by TuLiP in Paper C and Paper E have one certain output for each input. This means that they have one 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 E, that it may emerge behaviors that seems enforced, whereas they are actually not. Consider a formal model and specification that are synthesized into a tactical planner that exhibits a desired behavior that seems enforced but is not. Consider also that the tactical planner later must be augmented with a new specification 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 specification.

As Paper E illustrates, a maximally permissive supervisor *will* display such specification omissions while a reactive module *might* show them. As discussed in Section 4.5, it is difficult to understand what properties all the interactions between the specifications result in. Maximal 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 thesis 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 paths, 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 2.3). With $\square\lozenge$ it is therefore possible to specify that a system *shall* reach a certain state, and the synthesized reactive module *guarantees* that the state is eventually reached. Supremica, on the other hand, is 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 words for paths that do not include marked states (see Section 2.6). What this means for the modeling semantics in SCT is that a system *may* reach a marked state, and the synthesized supervisor must *allow* the marked state to be reached.

The specifications of Supremica cannot force 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 discusses 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 will then generate an empty reactive module.

Paper C outlines a method to, in a sense, force cycles in Supremica. 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). Though this does not produce cycles that are repeated forever, it makes sure that, if the plant so chooses, there is always the possibility for another cycle.

For instance, consider the automaton in Fig. 4.1. If the event e_2 is controllable, then a supervisor is allowed to disable e_2 and prevent further transitions from b to a. However, if e_1 is made uncontrollable, then a supervisor cannot

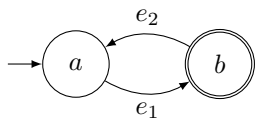

Figure 4.1: A simple example of an automaton with a cycle. If e_2 is uncontrollable, then a supervisor cannot prevent cycles to progress.

prevent arbitrary long cycles through a and b.

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 formulas 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

Discrete Modeling and Automatic Abstractions

Vehicles move in continuous time and space, but tactical planners make decisions at discrete time instances, and their available decisions may be from a discrete set of actions. Such systems, where a physical system and a digital computer affect and interact with each other, are often referred to as *Cyber-physical systems* [88]. The physical system has an infinite number of states and is governed by continuous dynamics in continuous time, whereas the digital computer has a finite number of states and is governed by discrete transition relations and discrete events.

The methods described in Chapter 2 and Chapter 4 all expect a finite state model as input. For these formal methods to be usable to provide proof of correctness for tactical planners operating in a cyber-physical system context, the physical system must be modeled as a finite state transition system. Hence, the continuous time evolution of the physical system must be turned into a transition relation between states, and the infinite state space of the physical system must be represented as a finite set of states. In this thesis, representing a continuous time evolution with a transition relation is called *discretization*, and representing a large number of states with fewer, as in the latter case, is called *abstraction*. The original physical system is from here on

referred to as the *continuous* system, the discretized system is referred to as the *concrete* system, and the system obtained after abstraction is referred to as the *abstract* system.

For the abstract system to be usable, its relation to the concrete system must fulfill properties such that verification or synthesis on the abstract system also carry some meaning in terms of the concrete system. What these properties are depend on the applied methods and requirements. As an example, it is easy to correctly abstract many systems to a transition system with one state and one transition that forms a self-loop that starts and ends in the same state. This abstraction may correctly represent the possible behaviors of the concrete system, albeit on a very abstract level, but the abstract system is not usable for any practical analysis of the concrete system. When using the abstract system for synthesis, a natural property to require on the relation between abstract and concrete system is that, if a planner fulfilling its specification is found for the abstract system, then a corresponding planner can also be found for the concrete system. For verification it is usually required that, if the concrete system system does not fulfill the specification, nor does the abstract system.

In general, from the point of view of the computational complexity of formal methods algorithms considered in this thesis, it is desirable to obtain an abstraction which is as coarse as possible. To be usable, however, the formalization of the requirements must still be sufficiently detailed. That is, the state set of the abstract system consists of as few states as possible, while the specification can still distinguish between good and bad behaviors.

This chapter details modeling considerations for manually modeling discrete abstract systems, and presents automatic methods to obtain discrete abstractions from infinite state systems.

5.1 Considerations for Manual Modeling

During manual modeling of finite transition systems, the abstraction can be done to different levels. For instance, Paper B and Paper C abstract a vehicle's longitudinal dynamics[1] in fine detail (or low abstraction), where the distance traveled is divided into many small steps. The discrete speed determines how

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

many steps the vehicle moves in each time step, just like the dice decide how many streets the players move in Monopoly[2].

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 vehicles shall 'keep a safe distance'. Both of these specifications are intuitive to state as they refer to end effects.

In the end, what is important for tactical planners is that they are safe, which 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 impractical for two reasons.

Firstly, the correctness of the planners is restricted to the certain model of the environment, which in Paper B means the specific lengths of the paths, and in Paper C means a circular road of a certain size and exactly one other vehicle. In practice, a vehicle is used in much more varied situations than what is modeled in Paper B and Paper C. Even if an automated vehicle has a very restricted ODD, 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 a weak dependence on detailed states. Since Paper B uses 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 vehicles' positions on the road. Removing the other vehicle or increasing the number of steps of the road after synthesis has been performed creates invalid states, so 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-space explosion problem, which means that the memory needed to store all states becomes too big to be tractable.

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

For any reasonably capable tactical planner the state-space explosion problem would be a serious obstacle with a low level of abstraction. The paths in Paper B has a maximal length of 1500 steps, and Spin [33] cannot handle much more. Supremica is close to the memory limit already when the road in Paper C is 100 steps long.

Paper D and Paper E 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 E, for instance, does not include position along a path or the vehicle's speed in the model of the environment. Instead, the model considers all positions along one path as one state, and the vehicle can either be stopped or driving. This higher level of abstraction leads to generic planners since the details of the paths in Paper E do 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 size of the state space. Then the synthesized planners become easier to inspect and interpret.

A downside of a high level of abstraction can be that each state captures a wide variety of behaviors. When constructing an argument in the safety case, this might put a large burden on the collection of evidence that should support the correctness of the abstract model. A high level of abstraction can therefore ease the effort for the synthesis or verification, while leaving so much effort on gathering evidence to justify the assumptions in the model that the gain of using formal methods is defeated.

5.2 Automatic Methods

Instead of manual modeling of finite state systems, it is possible to use automatic methods that divide the continuous state space into a finite number of blocks of a partition. One such abstraction method is *bisimulation* [89]. Bisimulation is guaranteed to find finite partitions for certain classes of systems [90], but in practice it is also useful for finding finite abstractions for systems of other classes. Bisimulation preserves all LTL properties [22], which makes it useful for automatically abstracting systems to be used in Model Checking or Reactive Synthesis with LTL specifications. The abstraction method

introduced in Paper F builds on bisimulation, so a short summary of bisimulation and some motivations are presented before the results of Paper F are discussed.

One common way of constructing the partition for the abstract system is to start with with a coarse partition and iteratively split blocks until the abstract system has the properties sought after.

Formally, a set $\Pi = \{P_1, P_2, \ldots, P_n\}$ is a *partition* of a set S if $P_i \neq \emptyset$ for all i, $P_i \cap P_j = \emptyset$ for all $i \neq j$, and $\bigcup_{1 \leq i \leq n} P_i = S$. That is, the elements of a partition Π are non-empty, disjoint, and together contain all the elements of S. The elements of a partition are called *blocks*. The union of an arbitrary number of blocks is called a *superblock*.

Let R be an equivalence relation on the set S. For an element $s \in S$, the *equivalence class* of s with respect to R is $[s]_R = \{s' \in S \mid (s, s') \in R\}$. The set of all the equivalence classes of R, also called the *quotient* of S modulo R, is $S/R = \{[s]_R \mid s \in S\}$. The quotient of S modulo R is a partition of S, and a partition Π induces an equivalence relation R if $S/R = \Pi$. A partition Π_1 is finer than a partition Π_2, and Π_2 is coarser than Π_1, if the induced equivalence relations R_1 and R_2 fulfill $R_1 \subseteq R_2$. For a more detailed and formal presentation of the notation in this chapter, see Section 2.

Bisimulation

Two states s_1 and s_2 in a transition system are considered *bisimilar* if the two following properties hold. First, they must be labeled with the same atomic propositions. Second, it must be the case that any transition from s_1 to state s_1' can be matched by a transition from s_2 to some state s_2', where s_1' and s_2' are themselves bisimilar. Consider for instance the transition system $G = \langle S, \Sigma, \delta, S^\circ, \mathrm{AP}, L \rangle$ in Fig. 5.1. The states a_1 and a_2 are bisimilar because they have the same label, and the only transition that can be taken from each of them leads to states that are bisimilar. On the other hand, the states b_1 and b_2 are not bisimilar, because b_1 has an outgoing transition to the state a_2 with label a and b_2 only has a transition to b_1. As a_2 and b_1 have different labels, they cannot be bisimilar, thus, so cannot b_1 and b_2.

More formally, a relation $R \subseteq S \times S$ is a *bisimulation relation* if for all pairs of states $(s_1, s_2) \in R$ it holds that: $L(s_1) = L(s_2)$; and, if there is some σ_1 and some state s_1' such that $(s_1, \sigma_1, s_1') \in \delta$, then there exists some σ_2 and some s_2' such that $(s_2, \sigma_2, s_2') \in \delta$ and $(s_1', s_2') \in R$; and vice versa. The

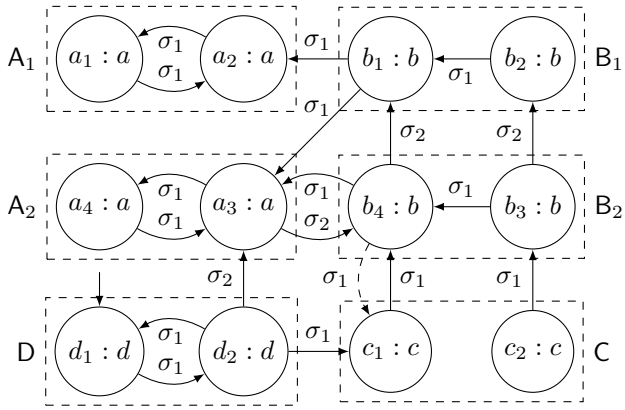

Figure 5.1: A transition system G adapted from Paper F. Circles represent states. The symbol before the colon in the text in the states is the state's name, and the symbol after the colon is its atomic proposition. The labels on the transitions are the actions. The capital letters are labels on the set of states represented by the dashed rectangles. The dashed transition from b_4 to c_1 is removed compared to Paper F.

coarsest partition Π that is closed under a bisimulation relation R is called the *bisimulation quotient*, and it is defined as $\Pi = S/R$ in Paper F.

The coarsest partition of the system G in Fig. 5.1 with respect to bisimulation consists of eleven blocks where $\{a_1, a_2\}$ is one block and the remaining states are in one block each. The abstract system, denoted by G/R, is constructed such that the abstract states are the blocks of the partition. Transitions are added between the states in the abstract system corresponding to the transitions between the concrete states. That is, if $(s_1, \sigma_1, s_1') \in \delta$ in G, then a transition is added in G/R between the block that contains s_1 and the block that contains s_1'. Hence, the abstract state $\{a_1, a_2\}$ in Fig. 5.1 will have a self-loop, and there will be a transition from the abstract state $\{b_1\}$ to $\{a_1, a_2\}$. The labeling function of the abstract system assigns labels according to the labels of the concrete states in the block. The label is always well defined since all the concrete states in every block must have the same label.

An algorithm to calculate the bisimulation quotient for a transition system is shown in Algorithm 1. It starts with a partition of S where the blocks consist of all states that have the same labels, as seen on lines 1 and 2 of Algorithm 1.

Algorithm 1: Coarsest bisimulation

Input: Transition system $G = \langle S, \Sigma, \delta, S^\circ, \mathrm{AP}, L \rangle$
Output: Coarsest bisimulation quotient

1 $R^0 \leftarrow \{(s_1, s_2) \in S \times S \mid L(s_1) = L(s_2)\}$;
2 $\Pi^0 \leftarrow S/R^0$;
3 $i \leftarrow 0$;
4 **while** *there exists a splitter* $T \in \mathcal{P}^i$ *of some block* P *in* Π^i **do**
5 $\quad P_0 \leftarrow P \cap \mathrm{Pre}(T)$;
6 $\quad P_1 \leftarrow P \setminus \mathrm{Pre}(T)$;
7 $\quad \Pi^{i+1} \leftarrow (\Pi^i \setminus P) \cup \{P_0, P_1\}$;
8 $\quad i \leftarrow i + 1$;
9 **end**
10 **return** Π^i;

Because bisimular states must have the same label, the partition $\mathcal{P}^0$ must be at least as coarse as the bisimulation quotient. In the while-loop, lines 4–9, the algorithm uses the one step predecessor operation $\mathrm{Pre}(\cdot)$ to determine whether blocks must be split further. The one step predecessor operation is defined for sets $T \subseteq S$ as

$$\mathrm{Pre}(T) = \{s \in S \mid \text{there exists } \sigma \in \Sigma \text{ and } t \in T \text{ such that } (s, \sigma, t) \in \delta\} \ .$$

It is the set of all concrete states that in one transition can reach a concrete state in the target set T. In line 4, a block T is a splitter of a block P if $P \cap \mathrm{Pre}(T) \neq \emptyset$ and $P \setminus \mathrm{Pre}(T) \neq \emptyset$. In Fig. 5.1, $\mathrm{Pre}(\mathsf{A}_1) = \{a_1, a_2, b_1\}$ is a splitter of $\mathsf{B}_1 = \{b_1, b_2\}$ because $\mathsf{B}_1 \cap \mathrm{Pre}(\mathsf{A}_1) = \{b_1\}$ and $\mathsf{B}_1 \setminus \mathrm{Pre}(\mathsf{A}_1) = \{b_2\}$. The blocks are split accordingly until none of the blocks need to be split further. More details of the algorithm can be found in the book by Baier and Katoen [22].

Linear discrete-time systems can be abstracted by a bisimulation algorithm by the use of linear operations on convex polytopes [91]. The one-step predecessor operator is defined in the same way, but the sets are represented as a combinations of linear constraints.

Consider the following linear discrete-time system:

$$x(k+1) = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 2 \\ 2 \end{bmatrix} u(k) \ , \tag{5.6}$$

where $x \in X \subseteq \mathbb{R}^2$ is the state, and $u \in U \subseteq \mathbb{R}$ is the control input. A transition system with an infinite number of states and transitions can be based on X as the state set, U as the set of actions, and (5.6) as the basis for the transition relation δ. See Section 2.1 in Paper F for a more general description.

The system (5.6) can describe the dynamics and the low-level controller of the vehicle in Paper B and Paper E, where $x = [d, v]^T$ is the vehicle's position and velocity along the path, respectively. The input signal u is the acceleration request, and is bounded to $U = [-0.5, 0.5]$. The vehicle must not pass the end of a path until a new one has been constructed. For the sake of example, assume that the current path p_1 starts at -4 and ends at 0, and that the next path, p_2, shall be planned from 0 to 4. This division of the state space X can be represented by labeling all the states with atomic propositions from $\mathrm{AP} = \{p_1, p_2\}$ according to whether the position d is positive or negative.

The result of running the bisimulation algorithm on (5.6) until there are no more splitters with an area larger than 0.01 area units is shown in Fig. 5.2. The state set X is represented by the thick black line, and each region within X represents one block in the final partition. However, since the algorithm is terminated when there still exist splitters, the coarsest bisimulation quotient is finer than the shown partition. The figure is meant only for illustrative purposes to show that the bisimulation quotient consists of many blocks.

The specification preventing the vehicle from driving off the end of a path in (B.11) in Paper B has the form $\Box(d > 0 \rightarrow q)$, where q is an atomic proposition representing whether the next path has been planned. With the propositions of the abstract system, this can be expressed as $\Box(p_2 \rightarrow q)$ because p_2 represents all positions greater than 0. A planner for the abstract system chooses at least one transition that is allowed in the current abstract state, and such a planner can be modeled manually and verified by Model Checking, or it can be synthesized by Reactive Synthesis.

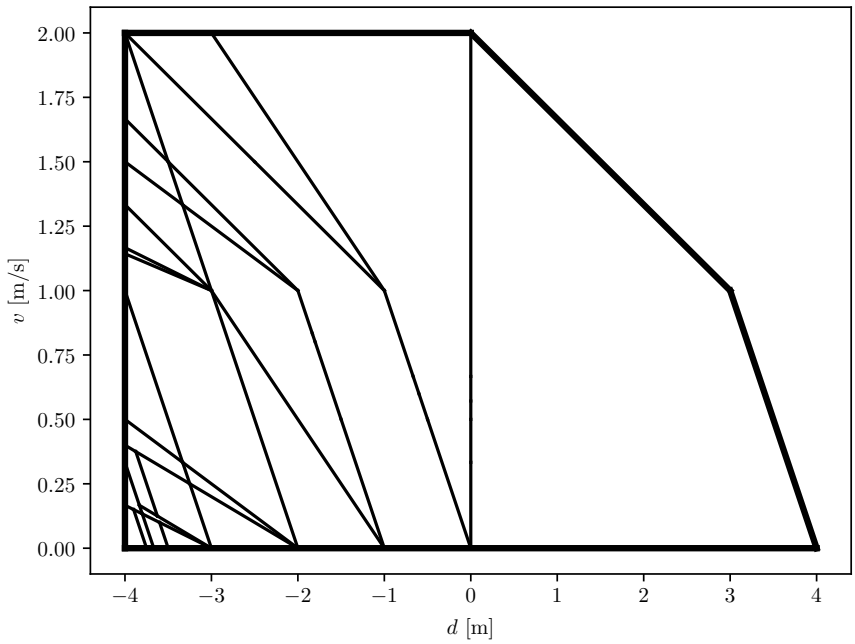

Figure 5.2: The resulting partition of (5.6) after Algorithm 1 has been run until all new splits result in blocks with an area less than 0.01 area units. The state set X is represented by the thick black line, and each region within X represents one block of the partition. The states labeled with p_2 are all in the large block to the right of 0 on the x-axis.

Robust Stutter Bisimulation

The feasibility of formal methods is highly dependent on the size of the state space, so it is desirable to construct abstract systems with as small state spaces as possible. As seen in Fig. 5.2, the bisimulation quotient for (5.6) consists of many blocks; too many for the abstraction to really be useful. Also, the usefulness of the coarsest bisimulation quotient of the transition system in Fig. 5.1 is debatable, since the abstract system only consists of one less state than the concrete system.

It can be observed in Fig. 5.1 that any path fragment starting in state d_1 is stutter trace equivalent to some path fragment starting in d_2, and vice versa.

That is, given a path fragment starting in d_1, it is always possible to find a path fragment starting in d_2 such that the order of the atomic propositions of the states visited by the path fragments are the same. Thus, d_1 and d_2 satisfy the same $\text{LTL}_{\setminus\circ}$ formulas. Hence, if the requirements to be verified or synthesized are free from the next operator, as is the case in Paper B, then d_1 and d_2 can be represented by the same abstract state.

To see why the removal of the next operator makes a difference, consider the LTL formula $\varphi \equiv \circ\, d$. The formula φ holds in d_1 of Fig. 5.1, but not in d_2. It holds in d_1 because if π_1 is a path fragment starting in d_1, then $\pi_1 = d_1 d_2 \cdots$ and $\mathsf{trace}(\pi_1) = dd\cdots$. By the definition of $\circ$, it holds that $dd\cdots \vDash \varphi$, and then it follows that $\pi_1 \vDash \varphi$, since $\circ\, d$ requires the second label (the *next* one) to be d. On the other hand, $\pi_2 = d_2 c_1 \cdots$ is a path fragment starting in d_2. Its trace is $\mathsf{trace}(\pi_2) = dc\cdots$, and it does not hold that $dc\cdots \vDash \varphi$. So $\pi_2 \nvDash \varphi$, and it follows that φ does not hold in d_2.

One relation that can consider d_1 and d_2 equivalent, and that preserves $\text{LTL}_{\setminus\circ}$, is the *divergent stutter bisimulation* relation [22]. A relation $R \subseteq S \times S$ is a divergent stutter bisimulation if, for all pairs of states $(s_1, s_2) \in R$ the following holds:

(i) $L(s_1) = L(s_2)$,

(ii) if there exists a σ_1 and a state s_1' such that $(s_1, \sigma_1, s_1') \in \delta$, then there exists some finite path fragment $\rho = s_2 u_1 u_2 \ldots u_n s_2'$ such that $(s_2, u_i) \in R$ for all $1 \leq i \leq n$, and $(s_1', s_2') \in R$,

(iii) if there exists an infinite path fragment $\pi_1 = s_1 v_1 v_2 \ldots$ such that $(s_1, v_i) \in R$ for all $i \geq 1$, then there exists some infinite path fragment $\pi_2 = s_2 u_1 u_2 \ldots$ such that $(s_2, u_j) \in R$ for all $j \geq 1$,

(iv) and vice versa.

In Fig. 5.1, the partition $\{\mathsf{A}_1, \mathsf{A}_2, \mathsf{B}_1 \cup \mathsf{B}_2, \mathsf{C}, \mathsf{D}\}$ is a divergent stutter bisimulation quotient.

An algorithm similar to Algorithm 1 can be used to compute divergent stutter bisimulation quotients. Instead of directly using the one-step predecessor operator Pre, the algorithm uses the $\text{PPre}(P, T)$ operator which is the set of states in P from which a state in T can be reached in finite number of steps without leaving P. With a slight abuse of notation, $\text{PPre}(P, T)$ essentially is all the states that satisfies $P\, \mathcal{U}\, T$. See Paper F for further details on this

non-standard notation of letting $\mathcal{U}$ operate on sets of states. Checking for splitters with $\mathrm{PPre}(P, T)$ ensures that condition (ii) is fulfilled. To ensure that condition (iii) is fulfilled, the algorithm essentially checks each block P for states that fulfill $\Box P$, and splits according to the same rules as in Algorithm 1. Fig. 5.3 shows the divergent stutter bisimulation quotient for the system (5.6) with the same state set as in Fig. 5.2.

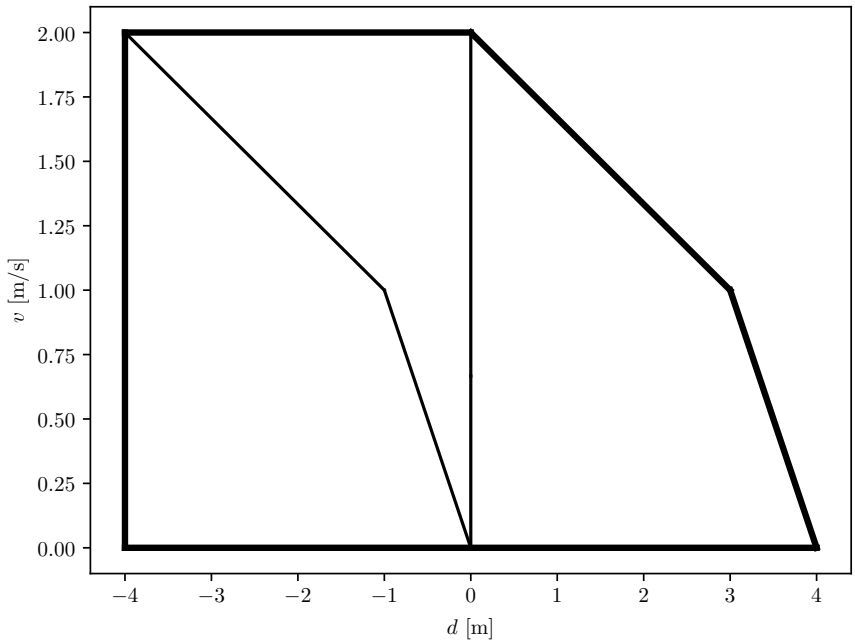

Figure 5.3: The divergent stutter bisimulation quotient of the system (5.6).

As can be seen in Fig. 5.3, the partition is much coarser than in Fig. 5.2. Given that none of the specifications in Paper B use the next operator, the satisfiability of them is not affected by the coarser abstraction. Seemingly, the reduction in state set size is obtained with little loss. However, the fewer abstract states are obtained by sacrificing structure, and the price being paid is the temporal meaning of the transitions in the abstract system. Taking one transition in an abstract system constructed from the bisimulation quotient corresponds to one transition in the concrete system, which means that "time"

passes with the same rate in both systems. For abstractions constructed from the divergent stutter bisimulation quotient, on the other hand, one transition in the abstract system can correspond to an arbitrary number of transitions in the concrete system. This number is also not fixed; one abstract transition can correspond to one concrete transition in one time instance, and it can correspond to 100 concrete transitions in another time instance. In essence, one transition in the abstract system might correspond to a millisecond, or it could correspond to an hour. The non-correspondence of number of transitions also means that it is non-trivial to construct a planner for the concrete system based on a planner for the abstract system, but it is known through explicit construction that a planner enforcing an $\text{LTL}_{\setminus\circ}$ formula φ on the concrete system exists iff there exists a planner enforcing φ on the abstract system [92].

When bisimulation or divergent stutter bisimulation are used to construct an abstract system for the purpose of designing or synthesizing planners, then, in general, the concrete system must be deterministic [92], [93]. This is exemplified by the block $\mathsf{B}_1 \cup \mathsf{B}_2$ in Fig. 5.1, where the action σ_1 leads to a non-deterministic transition from state b_1. As can be seen, any path fragment that starts in b_1 has a stutter trace equivalent path fragment starting in b_4, and vice versa. However, when a transition system is controlled by a planner, it is done so through the actions in Σ. Again, see Section 2.1 in Paper F why this is the case, and see Def. 8 on page F9 for a formal definition of how the planner (called a controller in Paper F) interacts with the system. From a planner's perspective, the $\text{LTL}_{\setminus\circ}$ formulas that can be *enforced* from b_1 and b_4 are not the same.

For instance, from b_4, a planner that always chooses σ_1 in b_4 and σ_2 in a_3 will only allow the path fragment $\pi_1 = b_4 a_3 b_4 a_3 \cdots$ which has the trace $\mathsf{trace}(\pi_1) = baba\cdots$. However, in b_1 a planner must choose σ_1, and then in one step the system may end up in either a_2 or a_3; which one is determined by an external agent and the choice is not known a priori by the planner. Hence, from b_1 the path fragment $\pi_2 = b_1 a_2 a_1 a_2 a_1 \cdots$ with trace $\mathsf{trace}(\pi_2) = baa\cdots$ is always a possibility. Since $\mathsf{trace}(\pi_1) = baba\cdots \vDash \square\lozenge b$, and since π_1 is the only path fragment allowed from b_4, it follows that $\square\lozenge b$ can be enforced from b_4. However, as $\mathsf{trace}(\pi_2) = baa\cdots \nvDash \square\lozenge b$, and as π_2 is always allowed from b_1, it follows that $\square\lozenge b$ cannot be enforced from b_1. Hence, b_1 and b_4 are not equivalent from a planner's perspective and should not be in the same block of a quotient.

Paper F presents a relation called *robust stutter bisimulation* that addresses abstraction and control of non-deterministic transition systems. It is shown that an $\text{LTL}_{\setminus\circ}$ formula φ can be enforced on the abstract system if and only if φ can be enforced on the concrete system. Algorithm 2 in Paper F computes the coarsest robust stutter bisimulation quotient, and an explicit construction method is presented for how a concrete planner can be constructed from an abstract planner.

The robust stutter bisimulation is useful, for instance, when the system dynamics include disturbances. Consider adding a disturbance to the discrete-time system of (5.6):

$$x(k+1) = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 2 \\ 2 \end{bmatrix} u(k) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} w(k) \ , \tag{5.7}$$

where $w \in W = [-0.25, 0]$ is additive noise. The robust stutter bisimulation quotient of (5.7) is shown in Fig. 5.4. The disturbance causes three small blocks to show up, since from the states in those blocks due to the disturbance it is not possible to force the system into the big middle block.

The robust stutter bisimulation is not only relevant for discrete-time systems subjected to additive disturbances. As stated above, an abstract system based on the bisimulation quotient matches temporally with the underlying concrete system. Recall from Section 5.2 that p_1 and p_2 refer to the planned paths, and that q holds if p_2 has been planned successfully. If the validity of the path p_2 expires after some time, then this dynamic can be modeled in a separate non-deterministic transition system Q where q does not hold after a certain number of transitions. Since the transitions in the bisimulation-based abstract system has a fixed temporal meaning, a product system of the abstract system and Q can be used as a model for synthesis.

However, a similar approach does not work if the abstraction is based on divergent stutter bisimulation. When the product is constructed, each transition in the abstract system is matched with some transition in Q. The transition in Q has a fixed time duration, but since the abstract transition is matched by an arbitrary number of concrete transitions, the abstract transition has a variable time duration. Hence, a transition that takes an hour in the abstract system might be matched to a transition in Q which takes one second.

The solution to this issue can be to construct the product from the concrete system and Q, and then use robust stutter bisimulation to abstract the product system.

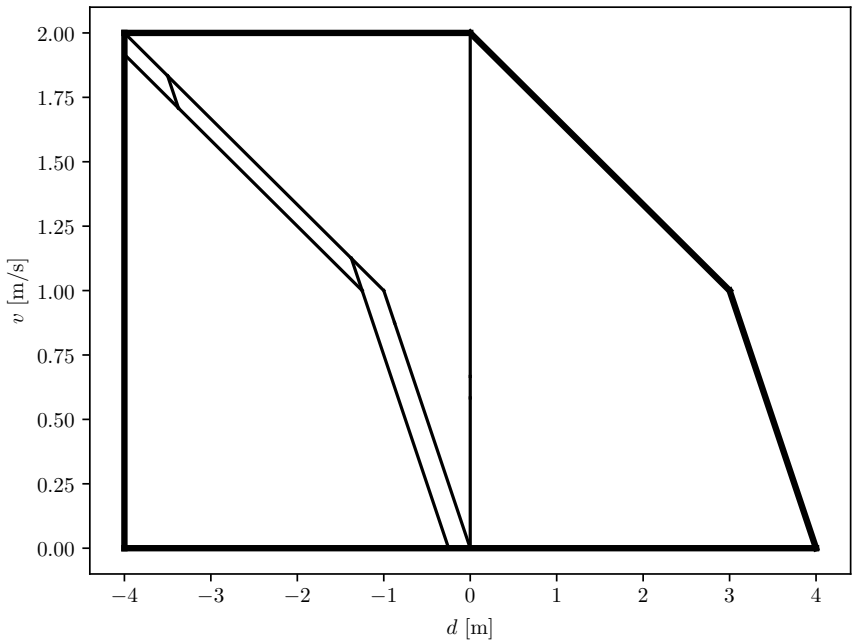

Figure 5.4: The robust bisimulation quotient of the system (5.7).

Further Studies

There are several opportunities when it comes to further studies on robust stutter bisimulation. Paper F shows an example where a robot navigates a maze, where the robot is modeled by a linear system with inputs and additive disturbances. An implementation of Algorithm 2 in Paper F was used to automatically compute the partition shown in Fig. 5 of Paper F. To understand the limitations and benefits of robust stutter bisimulation to a greater extent, it needs to be evaluated on more and different problem instances. Additionally, more research is needed on more efficient implementations. One thing in particular to investigate is whether the efficiency is affected by the order in which blocks and superblocks are evaluated for being splitters.

Furthermore, Algorithm 2 was implemented only to compute robust stutter bisimulations for linear systems with additive noise. It would be useful to extend the implementation to also compute the robust stutter bisimulation for

systems of the type in (5.6) with disturbance in the form of a finite transition system.

It might also be interesting to investigate how to handle measurement noise in robust stutter bisimulation.

As is shown in Paper F, robust stutter bisimulation preserves the existence of planners enforcing $\text{LTL}_{\setminus\circ}$ formulas. However, for some systems there are coarser abstractions that also preserve the existence of planners enforcing $\text{LTL}_{\setminus\circ}$ formulas. Consider for instance the transition system shown in Fig. 5.5. In this transition system, states a_2 and a_3 are not robust stutter bisimilar, because there is a planner that can force the system to stay in a_2 forever, but there is no planner that can force the system to stay in a_3. The state a_1 is not robust stutter bisimilar to either a_2 or a_3, and the robust stutter bisimilar quotient becomes $\Pi_1 = \{\{a_1\}, \{a_2\}, \{a_3\}, \{b_1\}\}$.

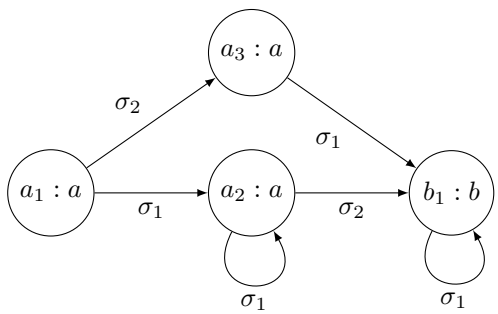

Figure 5.5: System with coarser partition than robust stutter bisimulation that still preserves the existence of controllers enforcing $\text{LTL}_{\setminus\circ}$ formulas.

Looking instead at the path fragments that can be enforced from a_1 and a_2, it is clear that for any path π_1 starting in a_1 there exists some path fragment π_2 starting in a_2 such that π_1 and π_2 are stutter trace equivalent. The other way around also holds, so it follows that the same $\text{LTL}_{\setminus\circ}$ formulas can be enforced from a_1 and a_2. Hence, $\Pi_2 = \{\{a_1, a_2\}, \{a_3\}, \{b_1\}\}$ is a partition that is coarser than the robust stutter bisimulation quotient, and that preserves the existence of controllers enforcing $\text{LTL}_{\setminus\circ}$ formulas.

This conclusion suggests two avenues of research. The first one is to answer the question "is there a more expressive logic which is a superset of $\text{LTL}_{\setminus\circ}$ that is preserved by Π_1?" A prime suspect to start investigating would then be

$\mathrm{CTL}^*_{\setminus\circ}$, a logic which is known to be preserved by divergent stutter bisimulation in verification contexts [22]; that is, when the abstraction is not intended to be used in synthesis.

The second research avenue is to implement an algorithm that can compute Π_2. Since all the requirements in Paper B can be formalized in $\mathrm{LTL}_{\setminus\circ}$, it would be interesting to investigate whether coarser partitions that preserve the existence of planners enforcing $\mathrm{LTL}_{\setminus\circ}$ formulas can be efficiently computed.

5.3 Abstractions in Safety Cases

It is argued in Paper A that to provide compelling safety evidence formal models must be used as requirements for the corresponding subsystems they model. For large transition systems, this could be a major obstacle to overcome during verification of the subsystem. Automatic abstractions might alleviate the effects of this obstacle by dividing the concerns. Instead of verifying that a manual model at a high level of abstraction correctly captures the behaviors of the subsystem, it is possible to verify the correctness of a model at a low level of abstraction and use the correctness proof of the abstraction to justify that the abstract system model is correct.

For instance, the models in Paper E is at a high level of abstraction, and evidence must be collected that supports the correctness of each state and transition. However, verifying that the system (5.6) is correct could be much easier than verifying the correctness of the models in Paper E, much because (5.6) allows for extrapolation. Once (5.6) is verified to be correct, then the correctness proofs in Paper F provide evidence that the abstract system is also correct.

CHAPTER 6

Conclusions

Formal methods have several benefits as tools to design tactical planners for automated vehicles. Given the requirements and an environment model, the planner can be guaranteed 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' available decisions may be from a discrete set of actions, and thus the requirements are natural to express in discrete states, which formal methods are particularly well suited for. 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 interactions will or must continue over time. To conclude and answer RQ 2, it seems that it is possible to apply formal methods for tactical-planner problems with the above mentioned characteristics.

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.

Examples in this thesis show that formal methods can be used to synthesize useful correct-by-construction tactical planners for automated vehicles. However, related to RQ 1, the application of formal synthesis to the generation of such planners has obstacles, mainly related to 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. Ascertaining that the synthesis result is reasonably correct could be done by visual inspection of planners with small number of states, or simulation to check that the right decisions are made. However, neither approach is 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 tens of thousands 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. In addition to answering RQ 1, the analysis of this obstacle is also relevant with respect to RQ 3. Safe planners must avoid collisions, so the capability to express such requirements are crucial. Two vehicles 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 vehicles. 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 vehicle; 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.

It follows that it is valuable to have methods that allow both detailed requirements and generic planners. This thesis proposes an automatic abstrac-

tion method called *robust stutter bisimulation* that attempts to combine these properties. It allows requirements to, for instance, express 'do not collide', while synthesizing the planner from the abstracted system. The robust stutter bisimulation provides one specific level of abstraction that is usable for tactical planners, giving a partial answer to RQ 3.

6.1 Future Work

The contributions and conclusions of this thesis are mainly on a technical and theoretical level. What has been investigated is the applicability of formal methods to model and specify the systems that are relevant for correct-by-construction tactical planners for automated vehicles. However, RQ 1 asks "what are the current limitations of formal synthesis for tactical planners for automated vehicles that hinders its *adoption* in the automotive industry." Clearly, identifying technical and theoretical obstacles is an important part in answering this question, but investigating engineering and organisational obstacles is just as pertinent to actually achieve adoption.

The overarching purpose of this thesis is guaranteeing safety of tactical planners. The correctness proof that formal methods provide is often touted as their greatest strength, but it is not established what impact the correctness proofs have on the real-life safety. An argument for how formal methods could provide compelling evidence for the safety of tactical planners and a proposal of how the safety case can be structured is put forth in Paper A. However, understanding the safety benefit of formal methods requires more than an argument. Actual effect on real-life safety must be investigated to understand the true benefits or drawbacks that comes from using formal methods to construct safety-critical tactical planners for automated vehicles.

Instead of attempting to use formal methods for creating correct-by-construction tactical planners, it might be worthwhile to use formal methods in the concept phase of projects to quickly construct planners that fulfill the current set of requirements. That way, the behaviors induced by the requirements could be evaluated before any implementation. It is also suggested by Paper A that formal methods could be put to use as a tool to assess the soundness and completeness of requirement break-downs. These two approaches have not been considered in this thesis, but could be relevant for future research.

CHAPTER 7

Summary of included papers

This chapter provides a summary of the included papers.

7.1 Paper A

> **Jonas Krook**, Yuvaraj Selvaraj, Wolfgang Ahrendt, Martin Fabian.
> A Formal-Methods Approach to Provide Evidence in Automated-Driving Safety Cases.
> Submitted to *IEEE Transactions on Intelligent Vehicles*, Oct. 2022.

Paper A contributes with an approach to structure a convincing safety argument with evidence from formal methods. The paper shows how formal methods can provide proof that tactical planners fulfill their requirements, and it demonstrates how formal methods can be used to close the gap between the complete system requirements and the broken down subsystem requirements. If the formal models are proven correct, the formal models are used as requirements on the subsystems. This structure of the safety argument can be used to alleviate the need for reviews and tests to ensure that the break-down is correct, thereby saving effort both in data collection and verification time.

7.2 Paper B

Jonas Krook, Lars Svensson, Yuchao Li, Lei Feng, Martin Fabian.
Design and Formal Verification of a Safe Stop Supervisor for an Automated Vehicle.
Published in *International Conference on Robotics and Automation*, pp. 5607–5613, May 2019. DOI: 10.1109/ICRA.2019.8793636.

In Paper B, formal verification is used to aid the design of a safety-critical tactical planner for an automated vehicle tasked with a transport mission. 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 during development 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.

7.3 Paper C

Zahra Ramezani, **Jonas Krook**, Zhennan Fei, Martin Fabian, Knut Åkesson.
Comparative Case Studies of Reactive Synthesis and Supervisory Control.
Published in *18th European Control Conference (ECC)*, pp. 1752–1759, Jun. 2019. DOI: 10.23919/ECC.2019.8795696.

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 or the other. For instance, it is demonstrated how shared resources affect the modeling, how cycles can be modeled, and how progress requirements are treated.

7.4 Paper D

Jonas Krook, Anton Zita, Roozbeh Kianfar, Sahar Mohajerani, Martin Fabian.
Modeling and Synthesis of the Lane Change Function of an Autonomous Vehicle.
Published in *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 133–138, Jul. 2018. 14th IFAC Workshop on Discrete Event Systems (WoDES). 2405-8963 DOI: 10.1016/j.ifacol.2018.06.291.

An earlier research study [26] applied formal verification and found issues in manually implemented code for an automotive application. Paper D 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.

7.5 Paper E

Jonas Krook, Roozbeh Kianfar, Martin Fabian.
Formal Synthesis of Safe Stop Tactical Planners for an Automated Vehicle.
Published in *IFAC-PapersOnLine*, vol. 53, no. 4, pp. 445–452, Nov. 2020. 15th IFAC Workshop on Discrete Event Systems (WoDES). 2405-8963 DOI: 10.1016/j.ifacol.2021.04.059.

The tactical planner designed in Paper B was formally verified to be correct. However, formal verification has some drawbacks, so Paper E 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 E 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 cause. Additionally, there is a conflict between the possibility to express detailed requirements while also generating generic planners.

7.6 Paper F

Jonas Krook, Robi Malik, Sahar Mohajerani, Martin Fabian.
Robust Stutter Bisimulation for Abstraction and Controller Synthesis with Disturbance.
Submitted to *Automatica*, Jun. 2022. DOI: 10.48550/arXiv.2205.13959.

The conclusion in Paper E, which is corroborated by Paper B and Paper C, that there is a conflict between detailed requirements and generic planners is addressed in Paper F. The problem is addressed by the introduction of the *robust stutter bisimulation* relation, which preserves the existence of robust planners for linear temporal logic formulas without the next operator. The robust stutter bisimulation is used to construct an abstract system with (hopefully) fewer states than the concrete system, and it is proven by explicit construction that there exists a controller enforcing a linear temporal formula for the abstract system if and only if there exists a corresponding controller for the concrete system. The results of the paper are useful for synthesizing controllers for systems subject to disturbances in the form of bounded noise or adversarial actions, and it is shown that the method works for a robot navigation example.

References

[1] *Regulation (EU) 2019/2144 of the European Parliament and of the Council on type-approval requirements for motor vehicles and their trailers, and systems, components and separate technical units intended for such vehicles, as regards their general safety and the protection of vehicle occupants and vulnerable road users*, Dec. 16, 2019. [Online]. Available: http://data.europa.eu/eli/reg/2019/2144/oj.

[2] SAE J3016_202104, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles," SAE Int., Tech. Rep., Apr. 30, 2021. DOI: 10.4271/J3016_202104.

[3] M. Hahm, "'Age in place': How self-driving cars will transform retirement," *Yahoo! Finance*, Sep. 7, 2017. [Online]. Available: https://finance.yahoo.com/news/age-place-self-driving-cars-will-transform-retirement-173308278.html (visited on 2019-12-10).

[4] S. Szymkowski, "Lyft, Aptiv will supply rides to blind or visually impaired people," *The Car Connection*, Jul. 13, 2019. [Online]. Available: https://www.thecarconnection.com/news/1124002_lyft-aptiv-will-supply-rides-to-blind-or-visually-impaired-people (visited on 2019-12-10).

[5] "U.S. partnership brings self-driving technology to the blind," *Intelligent Transport*, Jul. 11, 2019. [Online]. Available: https://www.intelligenttransport.com/transport-news/83700/us-partnership-brings-self-driving-technology-to-the-blind/ (visited on 2019-12-10).

[6] "Driverless cars tested by blind veterans," *ITV*, Jul. 13, 2019. [Online]. Available: https://www.itv.com/news/meridian/2019-07-13/driverless-cars-tested-by-blind-veterans/ (visited on 2019-12-10).

[7] A. Eugensson, M. Brännström, D. Frasher, M. Rothoff, S. Solyom, and A. Robertsson, "Environmental, safety, legal and societal implications of autonomous driving systems," in *Int. Tech. Conf. Enhanc. Saf. Veh.*, vol. 334, 2013.

[8] E. Lehtonen, F. Malin, T. Louw, Y. M. Lee, T. Itkonen, and S. Innamaa, "Why would people want to travel more with automated cars?" *Transp. Res. Part F: Traffic Psychol. Behav.*, vol. 89, pp. 143–154, 2022. DOI: 10.1016/j.trf.2022.06.014.

[9] S. Singh, "Critical reasons for crashes investigated in the National Motor Vehicle Crash Causation Survey," National Highway Traffic Safety Administration, Washington, DC, Tech. Rep., Feb. 2015.

[10] D. Zipper, "The deadly myth that human error causes most car crashes," *The Atlantic*, Nov. 26, 2021. [Online]. Available: https://www.theatlantic.com/ideas/archive/2021/11/deadly-myth-human-error-causes-most-car-crashes/620808/ (visited on 2022-07-08).

[11] P. Koopman, *A reality check on the 94 percent human error statistic for automated cars*, Jun. 5, 2018. [Online]. Available: http://safeautonomy.blogspot.com/2018/06/a-reality-check-on-94-percent-human.html (visited on 2022-07-08).

[12] N. Lubbe, H. Jeppsson, A. Ranjbar, J. Fredriksson, J. Bärgman, and M. Östling, "Predicted road traffic fatalities in germany: The potential and limitations of vehicle safety technologies from passive safety to highly automated driving," in *Int. Res. Counc. Biomech. Inj.*, Sep. 2018, pp. 17–52.

[13] 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. DOI: 10.7249/RR1478.

[14] S. Abuelsamid, "Transition to autonomous cars will take longer than you think, Waymo CEO tells governors," *Forbes*, Jul. 20, 2018. [Online]. Available: https://www.forbes.com/sites/samabuelsamid/2018/07/20/waymo-ceo-tells-governors-av-time-will-be-longer-than-you-think/ (visited on 2019-12-10).

[15] P. Koopman, A. Kane, and J. Black, "Credible autonomy safety argumentation," in *27th Saf.-Crit. Syst. Symp.*, Bristol, UK, Feb. 2019. [Online]. Available: https://users.ece.cmu.edu/~koopman/pubs/Koopman19_SSS_CredibleSafetyArgumentation.pdf.

[16] D. Åsljung, J. Nilsson, and J. Fredriksson, "Using extreme value theory for vehicle level safety validation and implications for autonomous vehicles," *IEEE Trans. Intell. Veh.*, vol. 2, no. 4, pp. 288–297, 2017. DOI: 10.1109/TIV.2017.2768219.

[17] 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. DOI: 10.1007/978-1-4613-2173-6_19.

[18] K. J. Åström and R. M. Murray, *Feedback Systes: An Introduction for Scientists and Engineers*. Princeton University Press, 2008.

[19] M. Hoss, M. Scholtes, and L. Eckstein, "A review of testing object-based environment perception for safe automated driving," *Automotive Innovation*, vol. 5, no. 3, pp. 223–250, Aug. 1, 2022. DOI: 10.1007/s42154-021-00172-y.

[20] L. Jing and Y. Tian, "Self-supervised visual feature learning with deep neural networks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 11, pp. 4037–4058, 2021. DOI: 10.1109/TPAMI.2020.2992393.

[21] ISO 26262:2018, "Road vehicles – functional safety," ISO, Tech. Rep., Dec. 2018.

[22] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[23] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd. New York, NY, USA: Springer Science & Business Media, 2008.

[24] O. Kupferman, P. Madhusudan, P. S. Thiagarajan, and M. Y. Vardi, "Open systems in reactive environments: Control and synthesis," in *CONCUR 2000 — Concurr. Theory*, ser. LNCS, vol. 1877, Springer, Berlin, Heidelberg, 2000, pp. 92–107. DOI: 10.1007/3-540-44618-4_9.

[25] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Apr. 2015. DOI: 10.1145/2699417.

[26] A. Zita, S. Mohajerani, and M. Fabian, "Application of formal verification to the lane change module of an autonomous vehicle," in *13th IEEE Conf. Autom. Sci. Eng.*, 2017. DOI: 10.1109/COASE.2017.8256223.

[27] 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 Eur. Congr. Embed. Real Time Softw. Syst.*, Toulouse, France, Jan. 2018.

[28] NASA, *What is formal methods?* Apr. 10, 2016. [Online]. Available: https://shemesh.larc.nasa.gov/fm/fm-what.html (visited on 2022-07-26).

[29] S. Yu, "Regular languages," in *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, G. Rozenberg and A. Salomaa, Eds. Springer Berlin Heidelberg, 1997, pp. 41–110, ISBN: 978-3-642-59136-5. DOI: 10.1007/978-3-642-59136-5_2.

[30] A. Pnueli, "The temporal logic of programs," in *18th Annual Symp. Found. Comput. Sci.*, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[31] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Fifteenth IFIP Int. Symp. Protoc. Specif., Test., Verif.*, 1996, pp. 3–18. DOI: 10.1007/978-0-387-34892-6_1.

[32] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," *Formal Methods Syst. Des.*, vol. 1, no. 2, pp. 275–288, Oct. 1, 1992. DOI: 10.1007/BF00121128.

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

[34] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer, "How to handle assumptions in synthesis," in *3rd Workshop Synth.*, ser. EPTCS, vol. 157, Jul. 2014, pp. 34–50. DOI: 10.4204/EPTCS.157.7.

[35] B. Finkbeiner, "Synthesis of reactive systems," in *Dependable Softw. Syst. Eng.*, vol. 45, 2016, pp. 72–98. DOI: 10.3233/978-1-61499-627-9-72.

[36] N. Piterman, "From nondeterministic Büchi and Streett automata to deterministic parity automata," *LMCS*, vol. 3, no. 3, Aug. 2007. DOI: 10.2168/LMCS-3(3:5)2007.

[37] P. J. Meyer, S. Sickert, and M. Luttenberger, "Strix: Explicit reactive synthesis strikes back!" In *Comput. Aided Verif.*, 2018, pp. 578–586. DOI: 10.1007/978-3-319-96145-3_31.

[38] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. 16th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1989, pp. 179–190. DOI: 10.1145/75277.75293.

[39] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of Reactive(1) designs," in *Verif., Model Checking, Abstr. Interpret.*, ser. LNCS, vol. 3855, Jan. 2006, pp. 364–380. DOI: 10.1007/11609773_24.

[40] 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 Conf. Control Appl.*, 2016, pp. 1030–1041. DOI: 10.1109/CCA.2016.7587949.

[41] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989. DOI: 10.1109/5.21072.

[42] 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.

[43] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992. DOI: 10.1145/136035.136043.

[44] Z. Fei, S. Miremadi, K. Åkesson, and B. Lennartson, "Efficient symbolic supervisor synthesis for extended finite automata," *IEEE Trans. Control Syst. Technol.*, vol. 22, no. 6, pp. 2368–2375, 2014. DOI: 10.1109/TCST.2014.2303134.

[45] 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, Jul. 2017, 20th IFAC World Congr. DOI: 10.1016/j.ifacol.2017.08.427.

[46] Ministry of Defence, "Defence standard 00-56 part 1: Safety management requirements for defence systems," Tech. Rep., Feb. 28, 2017.

[47] T. Kelly and R. Weaver, "The goal structuring notation – a safety argument notation," in *Int. Conf. Depend. Syst. Netw.*, Workshop Assur. Cases, Jul. 2004.

[48] ISO/PAS 21448:2022, "Road vehicles – safety of the intended functionality," ISO, Tech. Rep., Jun. 2022.

[49] G. Rodrigues de Campos, R. Kianfar, and M. Brännström, "Precautionary safety for autonomous driving systems: Adapting driving policies to satisfy quantitative risk norms," in *IEEE Intell. Transp. Syst. Conf.*, 2021, pp. 645–652. DOI: 10.1109/ITSC48978.2021.9564879.

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

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

[52] General Motors. "General Motors self-driving safety report." (2018), [Online]. Available: http://www.gm.com/mol/selfdriving.html (visited on 2019-12-10).

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

[54] Waymo. "Waymo safety report." (Feb. 2021), [Online]. Available: https://waymo.com/safetyreport/ (visited on 2022-07-20).

[55] P. LeBeau, "Waymo hits 10 millionth mile, prepares for public ride hailing," *CNBC*, Oct. 10, 2018. [Online]. Available: https://www.cnbc.com/2018/10/10/waymo-hits-10-millionth-mile-prepares-for-public-ride-hailing.html (visited on 2019-12-10).

[56] @Tesla. "As of today Tesla owners have driven 1 billion(!) miles with autopilot engaged." (Nov. 28, 2018), [Online]. Available: https://twitter.com/Tesla/status/1067810392322109441 (visited on 2019-12-10).

[57] F. Lambert, "Tesla drops a bunch of new Autopilot data, 3 billion miles and more," *Electrek*, Apr. 22, 2020. [Online]. Available: https://electrek.co/2020/04/22/tesla-autopilot-data-3-billion-miles/ (visited on 2022-07-20).

[58] I. Khrennikov, "Russia's Yandex joins the self-driving car million-mile club," *Bloomberg*, Oct. 17, 2019. [Online]. Available: https://www.bloomberg.com/news/articles/2019-10-17/russia-s-yandex-joins-the-self-driving-car-million-mile-club (visited on 2019-12-10).

[59] Yandex Self-Driving Team, "10 million kilometers in challenging conditions," *Medium*, Mar. 18, 2021. [Online]. Available: https://medium.com/yandex-self-driving-car/10-million-kilometers-in-challenging-conditions-d1d0045e5df0 (visited on 2022-07-20).

[60] D. Silver, "Miles driven," *Medium*, Jan. 8, 2018. [Online]. Available: https://medium.com/self-driving-cars/miles-driven-677bda21b0f7 (visited on 2019-12-11).

[61] K. Wiggers, "Uber's 250 autonomous cars have driven 'millions' of miles and transported 'tens of thousands' of passengers," *VentureBeat*, Apr. 11, 2019. [Online]. Available: https://venturebeat.com/2019/04/11/ubers-250-autonomous-cars-have-driven-millions-of-miles-and-transported-tens-of-thousands-of-passengers/ (visited on 2019-12-11).

[62] B. Vlasic and N. E. Boudette, "Self-driving Tesla was involved in fatal crash, U.S. says," *The New York Times*, Jun. 30, 2016. [Online]. Available: https://www.nytimes.com/2016/07/01/business/self-driving-tesla-fatal-crash-investigation.html (visited on 2019-12-11).

[63] D. Wakabayashi, "Self-driving Uber car kills pedestrian in Arizona, where robots roam," *The New York Times*, Mar. 19, 2018. [Online]. Available: https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html (visited on 2019-12-11).

[64] D. Shepardson, "U.S. investigates California Tesla crash that killed motorcyclist," *Reuters*, Jul. 18, 2022. [Online]. Available: https://www.reuters.com/world/us/us-investigates-california-tesla-crash-that-killed-motorcyclist-2022-07-18/ (visited on 2022-07-25).

[65] R. Beggin and K. Hall, "Cruise sets 2020 mileage record for av testing in california," *Government Technology*, Feb. 12, 2021. [Online]. Available: https://www.govtech.com/fs/cruise-sets-2020-mileage-record-for-av-testing-in-california.html (visited on 2022-07-20).

[66] F. Warg, M. Skoglund, A. Thorsén, R. Johansson, M. Brännström, M. Gyllenhammar, and M. Sanfridson, "The quantitative risk norm - a proposed tailoring of HARA for ADS," in *50th Annual IEEE/IFIP Int. Conf. Depend. Syst. Netw. Workshops*, 2020, pp. 86–93. DOI: 10.1109/DSN-W50199.2020.00026.

[67] 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," *Transp. Res. Part F: Traffic Psychol. Behav.*, vol. 62, pp. 870–882, 2019. DOI: 10.1016/j.trf.2019.03.020.

[68] A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt," in *IEEE Int. Symp. Requir. Eng.*, 1995, pp. 194–203. DOI: 10.1109/ISRE.1995.512561.

[69] C. Bergenhem, R. Johansson, A. Söderberg, J. Nilsson, J. Tryggvesson, M. Törngren, and S. Ursing, "How to reach complete safety requirement refinement for autonomous vehicles," in *Crit. Automot. Appl.: Robust. Saf.*, Sep. 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01190734.

[70] B. Littlewood and L. Strigini, "Validation of ultra-high dependability for software-based systems," in *Predict. Depend. Comput. Syst.*, 1995, pp. 473–493. DOI: 10.1007/978-3-642-79789-7_27.

[71] A. Galloway, F. Iwu, J. McDermid, and I. Toyn, "On the formal development of safety-critical software," in *First IFIP Conf. Verif. Softw.: Theories, Tools, Exp. 2005*, ser. LNCS, vol. 4171, 2008, pp. 362–373. DOI: 10.1007/978-3-540-69149-5_39.

[72] J. Rushby, "Formalism in safety cases," in *Mak. Syst. Safer*, ser. SCSC, Saf.-Crit. Syst. Symp., 2010, pp. 3–17. DOI: 10.1007/978-1-84996-086-1_1.

[73] E. Denney, G. Pai, and J. Pohl, "Heterogeneous aviation safety cases: Integrating the formal and the non-formal," in *IEEE 17th Int. Conf. Eng. Complex Comput. Syst.*, 2012, pp. 199–208. DOI: 10.1109/ICECCS20050.2012.6299215.

[74] E. Denney and G. Pai, "Evidence arguments for using formal methods in software certification," in *2013 IEEE Int. Symp. Softw. Reliab. Eng. Workshops*, 2013, pp. 375–380. DOI: 10.1109/ISSREW.2013.6688924.

[75] I. Habli and T. Kelly, "A generic goal-based certification argument for the justification of formal analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 238, no. 4, pp. 27–39, Sep. 28, 2009, First Workshop Certif. Saf.-Crit. Softw. Control. Syst. (SafeCert 2008). DOI: https://doi.org/10.1016/j.entcs.2009.09.004.

[76] ACWG, "Goal structuring notation community standard," SCSC, Tech. Rep., May 2021. [Online]. Available: https://scsc.uk/r141C:1.

[77] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *FM 2011: Formal Methods*, ser. LNCS, 2011, pp. 42–56. DOI: 10.1007/978-3-642-21437-0_6.

[78] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Synthesis of control protocols for autonomous systems," *Unmanned Syst.*, vol. 01, no. 01, pp. 21–39, 2013. DOI: 10.1142/S2301385013500027.

[79] F. Reijnen, M. Goorden, J. van de Mortel-Fronczak, and J. Rooda, "Supervisory control synthesis for a waterway lock," in *IEEE Annual Conf. Control Technology Appl.*, Aug. 2017, pp. 1562–1568. DOI: 10.1109/CCTA.2017.8062679.

[80] 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 Trans. Intell. Transp. Syst.*, vol. 19, no. 2, pp. 533–544, Feb. 2018. DOI: 10.1109/TITS.2017.2776354.

[81] Y. Tatsumoto, M. Shiraishi, and K. Cai, "Application of supervisory control theory with warehouse automation case study," *SYSTEMS, CONTROL AND INFORMATION*, vol. 62, no. 6, pp. 203–208, 2018. DOI: 10.11509/isciesci.62.6_203.

[82] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi, "Supervisory control and reactive synthesis: A comparative introduction," *Discrete Event Dyn. Syst.*, vol. 27, no. 2, pp. 209–260, Jun. 2017. DOI: 10.1007/s10626-015-0223-0.

[83] A.-K. Schmuck, T. Moor, and R. Majumdar, "On the relation between reactive synthesis and supervisory control of non-terminating processes," *Discrete Event Dyn. Syst.*, vol. 30, pp. 81–124, Mar. 2020. DOI: 10.1007/s10626-019-00299-5.

[84] A.-K. Schmuck, T. Moor, and K. W. Schmidt, "A reactive synthesis approach to supervisory control of terminating processes," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 2149–2156, 2020, 21st IFAC World Congr., ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.2541.

[85] D. Gunning and D. W. Aha, "DARPA's explainable artificial intelligence (XAI) program," *AI Magazine*, vol. 40, no. 2, pp. 44–58, Jun. 2019. DOI: 10.1609/aimag.v40i2.2850.

[86] R. Schmelzer, "Understanding explainable AI," *Forbes*, Jul. 23, 2019. [Online]. Available: https://www.forbes.com/sites/cognitiveworld/2019/07/23/understanding-explainable-ai/ (visited on 2019-12-11).

[87] R. Kumar and V. K. Garg, "Optimal supervisory control of discrete event dynamical systems," *SIAM J. Control Optim.*, vol. 33, no. 2, pp. 419–439, 1995. DOI: 10.1137/S0363012992235183.

[88] E. A. Lee, "The past, present and future of cyber-physical systems: A focus on models," *Sens.*, vol. 15, pp. 4837–4869, 3 2015. DOI: 10.3390/s150304837.

[89] R. Milner, *Communication and concurrency* (SCS). Prentice-Hall, 1989.

[90] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas, "Discrete abstractions of hybrid systems," *Proc. IEEE*, vol. 88, no. 7, pp. 971–984, 2000. DOI: 10.1109/5.871304.

[91] P. Nilsson, N. Özay, U. Topcu, and R. M. Murray, "Temporal logic control of switched affine systems with an application in fuel balancing," in *Am. Control Conf.*, Jun. 2012, pp. 5302–5309. DOI: 10.1109/ACC.2012.6315141.

[92] S. Mohajerani, R. Malik, A. Wintenberg, S. Lafortune, and N. Ozay, "Divergent stutter bisimulation abstraction for controller synthesis with linear temporal logic specifications," *Automatica*, vol. 130, no. 109723, Aug. 2021. DOI: 10.1016/j.automatica.2021.109723.

[93] G. Pola and P. Tabuada, "Symbolic models for nonlinear control systems: Alternating approximate bisimulations," *SIAM J. Control Optim.*, vol. 48, no. 2, pp. 719–733, 2009. DOI: 10.1137/070698580.