THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

On Provably Correct Decision-Making for Automated Driving

Yuvaraj Selvaraj

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2020

On Provably Correct Decision-Making for Automated Driving

Yuvaraj Selvaraj

This thesis has been prepared using LaTeX.

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

Printed by Chalmers Reproservice
Gothenburg, Sweden, August 2020

Abstract

The introduction of driving automation in road vehicles can potentially reduce road traffic crashes and significantly improve road safety. Automation in road vehicles also brings several other benefits such as the possibility to provide independent mobility for people who cannot and/or should not drive. Many different hardware and software components (e.g. sensing, decision-making, actuation, and control) interact to solve the autonomous driving task. Correctness of such automated driving systems is crucial as incorrect behaviour may have catastrophic consequences.

Autonomous vehicles operate in complex and dynamic environments, which requires decision-making and planning at different levels. The aim of such decision-making components in these systems is to make safe decisions at all times. The challenge of safety verification of these systems is crucial for the commercial deployment of full autonomy in vehicles. Testing for safety is expensive, impractical, and can never guarantee the absence of errors. In contrast, *formal methods*, which are techniques that use rigorous mathematical models to build hardware and software systems can provide a mathematical proof of the correctness of the system.

The focus of this thesis is to address some of the challenges in the safety verification of decision-making in automated driving systems. A central question here is how to establish formal verification as an efficient tool for automated driving software development.

A key finding is the need for an integrated formal approach to prove correctness and to provide a complete safety argument. This thesis provides insights into how three different formal verification approaches, namely supervisory control theory, model checking, and deductive verification differ in their application to automated driving and identifies the challenges associated with each method. It identifies the need for the introduction of more rigour in the requirement refinement process and presents one possible solution by using a formal model-based safety analysis approach. To address challenges in the manual modelling process, a possible solution by automatically learning formal models directly from code is proposed.

Keywords: Automated driving, formal methods, formal verification, decision-making, supervisory control theory, model checking, deductive verification, hybrid systems.

List of Publications

This thesis is based on the following publications:

[A] **Yuvaraj Selvaraj**, Wolfgang Ahrendt, Martin Fabian, "Verification of Decision Making Software in an Autonomous Vehicle: An Industrial Case Study". Larsen K., Willemse T. (eds): FMICS 2019. LNCS 11687, pp. 143–159, 2019.

[B] **Yuvaraj Selvaraj**, Zhennan Fei, Martin Fabian, "Supervisory Control Theory in System Safety Analysis". A.Casimiro et al. (Eds.): SAFECOMP 2020 Workshops, LNCS 12235, pp. 9-22, 2020.

[C] **Yuvaraj Selvaraj**, Ashfaq Farooqui, Ghazaleh Panahandeh, Martin Fabian, "Automatically Learning Formal Models: An Industrial Case from Autonomous Driving Development". ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion).

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

[D] R. Chandru, **Y. Selvaraj**, M. Brännström, R. Kianfar and N. Murgovski, "Safe autonomous lane changes in dense traffic". IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), Yokohama, 2017, pp. 1-6, doi: 10.1109/ITSC.2017.8317590.

Acknowledgments

I would like to thank Ali Hedayati for giving me the opportunity to pursue this Industrial PhD at Zenuity. It has been a journey full of learning and I am certain that the learning doesn't stop here.

I am privileged to have Prof. Martin Fabian and Prof. Wolfgang Ahrendt as my supervisors. The interdisciplinary discussions with both of them have immensely helped me shape the ideas in this thesis. I am thankful for the constant support, encouragement, and feedback they have provided me over the years.

I would like to thank all my colleagues at Zenuity and Chalmers. A special gratitude goes to Jonas Krook, Zhennan Fei, and Ghazaleh Panahandeh for all the discussions and feedback.

Finally, I would not be where I am today without my family who have continuously supported me in everything I do. This thesis would not have been possible without my wife, Lakshna. I am grateful for all the love and support you have given me. Thank you for having endured me through the good, and the bad.

Yuvaraj Selvaraj
Gothenburg, August 2020.

This work is supported by FFI, VINNOVA under grant number 2017-05519, *Automatically Assessing Correctness of Autonomous Vehicles–Auto-CAV*.

Acronyms

AD:	Automated Driving
ADAS:	Advanced Driver Assistance Systems
ASIL:	Automotive Safety Integrity Level
DES:	Discrete Event Systems
E/E:	Electrical and/or Electronic
EFSM:	Extended Finite State Machine
FSM:	Finite State Machine
HP:	Hybrid Program
LSM:	Lateral State Machine
SAE:	Society of Automotive Engineers
SCT:	Supervisory Control Theory
SOTIF:	Safety of the Intended Functionality
TLA:	Temporal Logic of Actions
WHO:	World Health Organization

Contents

Part I

Overview

CHAPTER 1

Introduction

The World Health Organization (WHO) reports [1] that approximately 1.35 million people die each year due to road traffic crashes which makes it a leading cause of human death globally. In addition, road traffic crashes are also a significant source of socio-economic losses amounting to $242 billion in the United States [2], and around 3% of the gross domestic product [1] in most countries. Studies show that 94% of serious crashes are due to human error [2], [3]. The introduction of driving automation in road vehicles can potentially reduce such crashes and significantly improve road safety. Automation in road vehicles also brings several other benefits. For example, it can help reduce drivers' stress by getting rid of the driving task and increase productivity as the driving time can be used efficiently for non-driving tasks. Another benefit is the possibility to provide independent mobility for people who cannot and/or should not drive [2], [3].

The Society of Automotive Engineers (SAE) categorizes driving automation systems into six discrete and mutually exclusive levels based on roles of the human driver and the driving automation system in relation to each other [4]. The amount of human supervision required is a critical part of this taxonomy and the levels extend from Level 0 (no automation) to Level 5 (full automation), as shown in Table 1.1.

<table>
<tr><td>SAE Level 0</td><td>SAE Level 1</td><td>SAE Level 2</td><td>SAE Level 3</td><td>SAE Level 4</td><td>SAE Level 5</td></tr>
<tr><td colspan="3">These are driver support features</td><td colspan="3">These are automated driving features</td></tr>
<tr><td colspan="3">Human driver responsible for driving whenever driver support features are engaged.</td><td colspan="3">Human driver is not driving when automated driving features are engaged.</td></tr>
<tr><td colspan="3">Constant human supervision is needed to maintain safety.</td><td>When the feature requests, human driver must drive.</td><td colspan="2">These automated driving features will not require the human driver to take over.</td></tr>
<tr><td>limited to providing warnings and momentary assistance</td><td>provide steering OR brake/acceleration support</td><td>provide steering AND brake/acceleration support</td><td colspan="2">These features can drive under limited operating conditions and will not operate until all requirements are met.</td><td>This feature can drive under all conditions.</td></tr>
<tr><td>automatic emergency braking, lane departure warning</td><td>lane centering OR adaptive cruise control</td><td>lane centering AND adaptive cruise control</td><td>traffic jam chauffeur</td><td>local driverless taxi (pedals, steering wheel may or may not be installed)</td><td>same as Level 4 but feature can drive everywhere in all conditions</td></tr>
</table>

Table 1.1: SAE levels of driving automation [5].

Currently, the level of automation in vehicles available for consumer purchase is either Level 1 or Level 2 [6]. Levels 1 and 2 are described as Advanced Driver Assistance Systems (ADAS) while Levels 3 to 5 are described as Automated Driving (AD) systems. With higher levels of automation, the complexity of the system increases and therefore several technical, business, and regulatory challenges arise in the commercial deployment of full autonomy in vehicles [7]. Among the technical challenges, sensor perception and decision-making have been strongly researched in both academia and industry

and significant progress has been made in the recent decade [8]–[11]. However, the challenge of safety verification and validation of these systems is yet to be solved [12]–[16].

The focus of this thesis is to address some of the challenges in the safety verification of decision-making in automated driving systems.

1.1 Industrial Challenges

An automated driving system consists of many software and hardware components interacting to solve different tasks, ranging from sensing, decision making, and planning to control and actuation. These tasks are achieved with the help of several electronic and/or electrical (E/E) subsystems that can be connected in many possible architectural designs. In this thesis, we consider an AD system architecture to consist of three subsystems as shown in Figure 1.1.

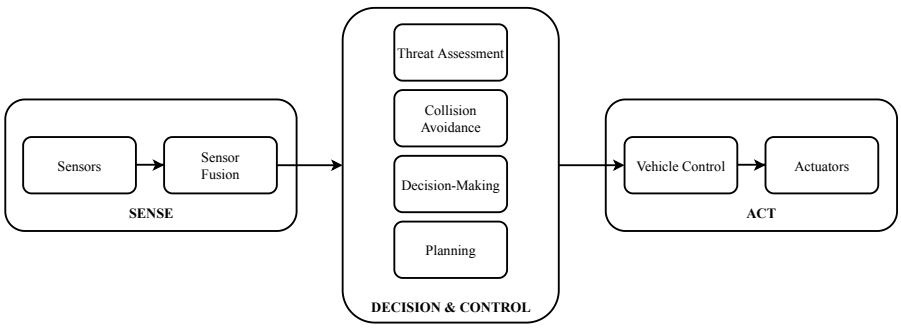


Figure 1.1: System Architecture

The SENSE subsystem includes sensors such as cameras, radars, gyros, etc., that perceive the autonomous vehicle's surroundings and provide information to the sensor fusion component. This component fuses all the information from the different sensors to provide a mathematical model of the traffic situation, such as the vehicle state, road geometry, distance to other vehicles, etc. The DECISION & CONTROL subsystem uses the fused information to decide when and how to act. Typical sub-tasks here are threat assessment, decision-making, path planning, etc. The decisions are then communicated to the ACT

sub-system in the form of, e.g., acceleration and steering commands. Finally, these decisions are executed by using, e.g. brake and/or steering actuators to control the vehicle.

Although human error is a major cause of many traffic accidents, the actual failure rate is remarkably low. For instance, in the United States, the fatality rate is 1.13 deaths per 100 million miles driven [12], [17]. This low rate presents a challenge as the failure rate of AD systems have to be at least better than human driving in order to increase overall traffic safety. Such a requirement presents difficulties in the development and verification of these automation systems. While human supervision is necessary to maintain safety in driver support features (levels 0-2 of Table 1.1), there needs to be a convincing safety argument for higher levels of automated driving features. Ensuring safety of AD systems is a multi-disciplinary challenge and several approaches have been researched and attempted to establish the correctness and subsequently prove the safety of AD systems [15], [16].

At present, the most widely used industrial approach to assure automotive software safety is *testing*. Testing can be done at various levels, such as software unit testing, model-based testing, complete vehicle on-road testing, and scenario-based virtual simulations. However, there are several challenges associated with a testing based approach to prove safety [13]. The most prominent one among them is that testing can never guarantee the absence of errors and is therefore infeasible to ensure completeness. Testing for safety is also very expensive and even impractical. Studies have shown [12] that it would be necessary to drive hundreds of thousands of miles, and sometimes even billions of miles to demonstrate that autonomous vehicles are better than humans.

The shortcomings in current industrial practice are further highlighted by the impact of defective software deployed in production vehicles. Several potentially dangerous software defects have been reported in the past that could result in catastrophes not due to driver error [18], [19]. These defects have resulted in hundreds of thousands of vehicle recalls as they pose an unreasonable safety risk. In addition, fatalities and crashes involving driving automation systems in Tesla [20], [21], Uber [22], and Google [23], [24] have warranted stricter safety arguments for successful deployment of autonomous vehicles at scale.

Formal methods, which are techniques that use rigorous mathematical models to build hardware and software systems can, in contrast to testing, provide

a mathematical proof of the correctness of the system. The strength of formal methods lie in the use of unambiguous formal logic and the possibility to provide a sufficiently complete safety argument. Formal methods consists of various diverse techniques that can be classified based on the proof approach used and the extent of formalisation used to prove the correctness. Of the different formal approaches, a particularly useful one that aids in achieving a higher level of safety assurance for automated driving systems is *formal verification*.

Given a formal model of a system and a *formal specification* of the intended behaviour of the system, formal verification is the act of proving or disproving the correctness of the system with respect to its specification. Formal verification has previously been used in safety-critical industries such as avionics [25], railway systems [26], and nuclear plants [27]. Though formal verification is shown to be useful in the automotive industry [28]–[30], it is not widely used as an efficient tool in the development process. This clearly points to the presence of unique challenges that hinder widespread industrial adoption of formal verification.

1.2 Research Questions

This thesis mainly concerns the question of how to establish formal verification as an efficient tool for automated driving software development. The thesis starts with the hypothesis that formal methods, especially formal verification, can be used to prove the correctness of AD systems and thereby provide sufficient evidence for strong safety arguments. While formal verification is shown to be beneficial in automotive applications [28]–[32], it is yet to become a standard tool in the development of automated driving software, most probably due to certain obstacles. The central focus of this thesis is to investigate these obstacles and explore how formal verification can be used to prove the correctness of decision-making subsystems in autonomous vehicles. Based on the included papers, this thesis aims to answer the following research questions:

RQ 1 What factors affect the application of formal verification to automated driving systems and what are the current challenges in existing methods?

RQ 2 How can we address the challenges in **RQ 1**, and how can the solutions be scaled?

RQ 3 How can we integrate the solutions to **RQ 2** in the day-to-day software development process?

The work of this thesis has been carried out as a part of an industrial PhD project at Zenuity AB.[1] The primary method employed to answer the research questions is in the form of industrial case studies, where the problem statements addressed have their roots in the industrial research and development of automated driving systems.

1.3 Scientific Contributions

The main scientific contributions in this thesis are documented in the included papers. Paper A identifies several challenges in applying formal verification to prove correctness of AD systems. It provides insights into how different formal verification methods differ with respect to the choice of formalism and identifies the need for an integrated approach to provide a sufficiently strong safety argument. Two of the challenges identified in Paper A are further investigated in Paper B and Paper C. Paper B demonstrates how formal methods can be used to perform model-based safety analysis and this thesis discusses how such an approach can be used to write better requirements for AD systems. Furthermore, the case studies in this thesis strengthens the fact that manual effort needed in formal verification is indeed an obstacle that impedes industrial adoption. Paper C investigates one possible approach to alleviate the problem of manually constructing formal models for decision-making systems.

1.4 Thesis Structure

The thesis is divided into two parts. Part I gives an introduction to automated driving, the challenges in proving correctness of AD, the necessary fundamentals, and a summary of scientific contributions from the included papers. Part II contains the papers. Part I consists of the following chapters:

[1] Zenuity AB recently changed its company identity to Ztwo company AB.

Chapter 1 - Introduction
This current chapter provides a brief introduction to automated driving, the research questions and the contributions of this thesis.

Chapter 2 - The Question of Proof
The second chapter puts into perspective how formal methods compare to the other current industrial approaches to provide credible safety argument for automated driving.

Chapter 3 - Formal Methods
The third chapter presents the necessary preliminaries and the fundamental concepts of different formal verification methods discussed in this thesis.

Chapter 4 - Provably Correct Decision-Making
This chapter presents insights obtained from the three papers towards provable correctness for decision-making systems and connects them in the context of this thesis.

Chapter 5 - Summary of Included Papers
This chapter provides a brief summary of the included papers and their contributions.

Chapter 6 - Concluding Remarks and Future Work
The final chapter concludes the first part of the thesis and presents ideas for future research.

CHAPTER 2

The Question of Proof

From Chapter 1, it is clear that correctness of AD systems is crucial for their safe and successful deployment at scale. While several approaches [12], [15], [16] have been researched and adopted to provide sufficient evidence for safe and correct behaviour of AD, they do have their own limitations. This chapter presents a brief overview of how formal methods compare to some of the other industrial approaches for safety verification of AD systems.

2.1 Conformance to Safety Standards

A familiar strategy for safety argumentation is to show evidence for conformance to industry-specific safety standards. For instance, many safety standards exist in the transportation industry to aid in the development of safety-critical software. The DO-178C standard [33] is used by various certification authorities in the aerospace industry to certify software for aerospace systems. Similarly, the railway industry uses the EN50128 (IEC 622279) [34] as one of the safety standards. For the automotive industry, the ISO 26262 functional safety standard [35] is the primary standard to address the safety of electrical and/or electronic (E/E) systems within road vehicles.

ISO 26262

ISO 26262 describes a functional safety framework for the development of safe E/E systems. Functional safety, as defined by the standard is the absence of unreasonable risks that could be caused by the malfunctioning behaviour of the E/E systems. The standard provides a reference for the automotive safety lifecycle and takes a risk-based approach to mitigate and/or avoid risks. ISO 26262 gives appropriate guidelines to achieve functional safety throughout the development process, which includes activities like requirement specification, design, implementation, verification and validation, etc.

The initiating task of the safety lifecycle is the *item* definition. The objective of the item definition phase is to develop a description of the system, or combination of systems, for which functional safety needs to be achieved and subsequently determine its functionality, boundary, interfaces, environmental conditions, assumptions, etc. The item definition phase is then followed by a hazard analysis and risk assessment phase where potential hazardous events of the item are identified and classified based on three factors:

- the *severity* of the potential harm caused by the hazardous event,
- the probability of *exposure* of the hazardous event, and
- the *controllability*, which estimates the ability to avoid the specific harm.

Together, the three parameters determine the Automotive Safety Integrity Level (ASIL). Four possible levels: ASIL A, ASIL B, ASIL C, and ASIL D are defined to categorize the hazardous events. ASIL A is the lowest safety integrity level and ASIL D is the highest. These levels are then used to obtain the safety goals that form the top-level safety requirements for the system. These safety goals are the key to achieving functional safety and the standard recommends best practices for the design, implementation, and verification of the item such that the safety goals are met.

In subsequent phases of the safety lifecycle, safety goals are broken down to obtain detailed safety requirements. These detailed requirements, derived as a result of the requirement refinement process, correspond to different phases in the design. For instance, the safety goals lead to implementation-independent functional safety requirements, which in turn are refined to obtain implementation specific technical safety requirements. Since the ASIL allotted to a

safety goal is inherited throughout the process, sufficient verification evidence for all the requirements is needed to comply with the safety goal.

Evidently, a higher ASIL demands more rigorous development processes compared to a lower ASIL. This fact is also reflected in the recommended best practices for the development of high ASIL components in the standard. For instance, while the use of semi-formal and formal methods (in addition to other methods) are recommended for higher ASIL components, the standard has no recommendation for or against their usage for lower ASIL components.

Several arguments [13], [15], [36] have been presented to show that ISO 26262 presents unique challenges and is sometimes even inadequate to completely demonstrate AD safety. A specific concern is the impact of the inherent assumptions that are made in the standard. As an example, in the ASIL analysis, especially the controllability parameter assumes a human driver to react and mitigate/avoid risks. While such an assumption might be relevant in ADAS features, it is fatal for AD. Another arguable point is the assumption of the V model [35], [37] in the reference safety lifecycle. The relevance of ASIL analysis and the recommended best practices in a development approach other than the V model is debatable. Such challenges have initiated the development of new standards [38], [39] specific to autonomous systems.

Other Relevant Standards

The recently published UL 4600 [38] standard addresses the safety processes for the evaluation of autonomous vehicles and other products without human supervision. The standard is intended to provide additional elements to address the needs for safety of full autonomy and is therefore expected to work with other existing standards. UL 4600 takes a claim-based safety case approach that includes structured arguments and evidence to support the fact that an item is acceptably safe for deployment. While UL 4600 provides guidance and recommended best practices to improve the completeness of the safety case, it does not mandate any specific development process (e.g., V model). Most importantly, UL 4600 highly recommends the use of formal methods as a verification and validation approach to provide sufficient evidence of acceptable safety.

The ISO/PAS 21448 [39] standard addresses the safety of systems that rely on sensing the environment, which is essential for AD. The ISO/PAS 21448 standard concerns the safety of the intended functionality (SOTIF) and is

complementary to the functional safety aspect of ISO 26262. SOTIF provides guidelines to achieve absence of unreasonable risks due to potential hazards caused by limitations in a system that is free from faults addressed in the ISO 26262 standard. Limitations that arise due to the inability of a function to have correct situational awareness and performance issues due to sensor variations are typical examples of risks dealt with in this standard.

Though these standards exist, conformance to any particular standard alone is not a guarantee of a 100% safe autonomous vehicle. However, conformance to such standards provides the necessary rigorous evidence to support an acceptably safe autonomous vehicle. Formal methods, by definition requires rigorous mathematical proofs and therefore meets the objective of such safety standards, wherever applied.

2.2 Testing, Simulation, and Miles Driven

As mentioned in Chapter 1, testing is a common quality assurance strategy within the automotive industry. Requirements based testing is an important aspect of the V model development framework in the reference safety lifecycle of ISO 26262. An approach only based on testing for AD safety includes many challenges and the most important ones are highlighted in this section.

The primary concern is that testing can only be used to find faults and can never guarantee their absence. Therefore, such an approach presents an incomplete safety argument. Testing can be done at various levels that range from software unit testing to complete vehicle testing. Driving field-test vehicles in real world conditions is one of the approaches undertaken to demonstrate acceptably safe autonomous vehicles. An important practical question here is how many test miles need to be driven? The answer is that it would require hundreds of millions of miles to statistically demonstrate the reliability of AD in terms of fatalities and injuries compared to human driving [12]. In addition to being expensive and potentially dangerous to public safety, such an approach is practically intractable.

One way to reduce the cost and scale of vehicle-level testing is to run the tests virtually by means of simulations in a wide range of scenarios representing real world conditions. While this approach overcomes some of the disadvantages of field testing, it brings a different set of unique challenges. For instance, the reliability of the simulation framework, selection of statis-

tically significant simulations, and the need for massive amounts of data to accurately represent real world environment are some of the challenges that have to be addressed [15], [16].

2.3 Formal Verification

Although formal methods, especially formal verification does not remove the need for testing, it can provide a higher level of safety assurance in the form of correctness proofs. In comparison to testing, which requires *dynamic* analysis (e.g. executing code, running software in a vehicle, etc.), formal verification techniques are *static* in nature. This makes it best suited to detect subtle errors in the early stages of development and potentially avoid catastrophic consequences associated with vehicle recalls [18], [19]. Formal verification can indeed provide sufficient rigour in the safety argument as required by different safety standards. Despite the advantages, the industrial adoption of formal verification in the automotive industry for AD development is below par. In order to ease their industrial adoption for safe AD development, the associated challenges need to be explored and possible solutions have to be investigated. In the subsequent chapters, this thesis throws some light on these aspects.

CHAPTER 3

Formal Methods

Admittedly, formal methods is an overarching term that includes a variety of tools and techniques. Currently, an online repository[1] lists more than 100 different formal notations and tools available for describing and reasoning about computer-based systems. There are many different existing taxonomies to classify formal methods. One way of classification is *lightweight* [40] versus what we in contrast can call *heavyweight* formal methods. This taxonomy is based on the ease of use and the extent of formalization in the development process. While the more common heavyweight approach uses full formalization and is more powerful, lightweight formal methods use partial (less than complete) formalizations and do not require deep expertise.

In addition to the above taxonomy, Almeida *et al.* [41] present other types of classification based on different factors. According to Almeida *et al.*, the characteristic aspect of formal methods is the ability to guarantee the behaviour of a given system with some rigorous approach. Therefore, the *specification*, which is a description of the intended behaviour, is at the core of formal methods. In this context, the different groups of formal methods can be classified into four types as shown in Table 3.1.

[1] https://formalmethods.wikia.org/wiki/Formal_methods

Classification	Approach	Example
Specify and Analyse	specification is formally defined and analysed using e.g. animation, execution, etc.	ASM, Z, VDM, CASL
Specify and Prove	specifications and models are formally defined to prove properties about them (**formal verification**)	model checking, SPARK, KeY
Specify and Derive	given a formal specification, obtain an implementation whose behaviour matches the specification (**correct-by-construction**)	supervisory control theory, program refinement
Specify and Transform	transform a specification to either approximate or enrich it	abstract interpretation

Table 3.1: Classification of formal methods [41]

In this thesis, the principal focus is on the safety verification of AD systems, and therefore we limit ourselves to the "Specify and Prove", i.e., the formal verification category of formal methods. As soon as we enter the realm of formal verification, a natural question that follows is: how do we specify, and how do we prove properties of the systems? Of course, there are multiple ways to specify and prove depending on the modelling formalism and the proof technique used. Traditionally, the choice of verification method for a computer-based system has been dependent on the nature of the system. For instance, model checking using SPIN [42] can be used to verify distributed asynchronous process systems, and the deductive verification tool KeY [43] can be used to verify Java programs.

In the case of AD systems, such a straightforward choice of the verification method is no longer possible due to the complexity involved. The sensor fusion component of the AD system consists of probabilistic algorithms and therefore probabilistic techniques such as the PRISM model checker [44] could be a potential choice. The vehicle control component is built on strong control theoretic and analytical aspects of continuous state systems and therefore necessitate tools that can handle these types of systems, such as the continuous reachability analysis toolbox [45]. The decision and control component includes various sub-components of different nature (discrete, continuous, hybrid, concurrent, etc.) and therefore multiple methods are suitable. As discussed in Chapter 4, a logical formalism that supports hybrid system verifi-

cation such as the differential dynamic logic [46], [47] is highly favourable in this regard. The use of machine learning algorithms in AD systems further complicates the verification process.

A variety of formal verification techniques have previously been used to reason about autonomous systems [48], [49]. In this thesis, we focus on three general methods shown in Table 3.2. The choice of these methods is primarily motivated by the following important factors:

- characteristic of the decision-making system in automated driving,[2]
- established proof of concept for verification of safety-critical systems,
- capability to handle industrial sized systems.

Approach	Technique	Tools
Control theoretic	supervisory control theory, invariant-set control theory	SUPREMICA, CORA
Model checking	explicit, symbolic, bounded	TLA$^+$, nuXmv, CBMC
Deductive verification	Hoare logic, dynamic logic, differential dynamic logic	SPARK, KeY, KeYmaera

Table 3.2: Examples of formal verification techniques

3.1 Supervisory Control Theory

The Supervisory Control Theory [50] (SCT) provides a framework for modelling, synthesis, and verification of reactive control functions for *discrete event systems* (DES), which are systems that occupy at each time instant a single *state* out of its many possible ones, and transit to another state on the occurrence of an *event*. SUPREMICA [51] is a tool that implements SCT based algorithms for the verification and synthesis of DES models. SUPREMICA has previously been shown to handle industrial sized systems [52] and also for the synthesis and verification of AD systems [30], [53].

[2] A brief description of decision-making in AD systems is presented in Chapter 4.

Given a DES model of a system to control, the *plant*, and a *specification* of the desired controlled behaviour, the SCT provides means to synthesize a *supervisor* that interacting with the plant in a *closed-loop* dynamically restricts the event generation of the plant such that the specification is satisfied. Though the original SCT focused on synthesising supervisors that by construction fulfil the desired properties, a dual problem of interest here is to, given a model of a plant and specification, verify whether the specification is fulfilled or not.

A DES modelling formalism appropriate in our context is finite-state machines extended with bounded discrete variables, with guards (logical expressions) over the variables and actions that assign values to the variables on the transitions [54].

Definition 1: *An Extended Finite State Machine (EFSM) is a tuple* $E = \langle \Sigma, V, L, \rightarrow, L^i, L^m \rangle$*, where* Σ *is a finite set of events,* V *is a finite set of bounded discrete variables,* L *is a finite set of locations,* $\rightarrow \subseteq L \times \Sigma \times G \times A \times L$ *is the conditional transition relation, where* G *and* A *are the respective sets of guards and actions,* $L^i \subseteq L$ *is the set of initial locations, and* $L^m \subseteq L$ *is the set of marked locations.*

The current state of an EFSM is given by its current location together with the current values of its variables. Thus, the state of an EFSM is not necessarily explicitly enumerated, but can be represented symbolically. This richer structure, though with equal expressive power, shows good modelling potential compared to ordinary finite state machines. The expression $l_0 \xrightarrow{\sigma:[g]a} l_1$ denotes a transition from location l_0 to l_1 labelled by event $\sigma \in \Sigma$, and with guard $g \in G$ and action $a \in A$. The transition is enabled when g evaluates to $\mathbf{T}$, and on its occurrence the EFSM changes location from l_0 to l_1, while a updates the values of some of the variables.

EFSMs naturally interact through shared variables, but they can also interact through shared events, which is modelled by *synchronous composition*, where common events occur simultaneously in all interacting EFSMs, or not at all, while non-shared events occur independently. By this interaction mechanism a supervisor restricts the event generation of the plant; if the supervisor has a specific event in its alphabet but has no enabled transition labelled by that event from its current state, then the closed-loop system cannot execute that event in the current global state. We denote the synchronous composition of two EFSMs E_1 and E_2 by $E_1 \parallel E_2$ [54]. As defined by [54], transitions

labelled by shared events but with mutually exclusive guards, or conflicting actions can never occur.

Nonblocking Verification

Given a set of EFSMs $\mathcal{E} = \{G_1, \ldots, G_n, K_1, \ldots, K_m\}$ where the components G_i $(i = 1, \ldots, n)$ represent the plant, and K_j $(j = 1, \ldots, m)$ represent the specification, we now want to determine whether the synchronous composition over all the components can from any reachable state always reach some marked state. The straightforward way to do this, called the *monolithic* approach, is intractable for all but the smallest systems, due to the combinatorial state-space explosion problem. Thus, more efficient approaches are needed.

One such approach that pushes the limit of what is tractable is the *abstraction-based compositional verification* [52], which has shown remarkable efficiency and manages to handle systems of industrially interesting sizes and complexity. It can be shown [52] that when $\mathcal{E}$ is blocking, this is due to some *conflict* between the components of $\mathcal{E}$. Thus, the approach of [52] employs *conflict-preserving abstractions* to iteratively remove redundancy and thus to keep the abstracted system size manageable. However, this approach eventually ends up converting the resulting abstracted EFSM system into ordinary finite-state machines, and then doing a monolithic verification of that. This then requires an efficient *explicit* verification algorithm, such as the one presented in [55].

3.2 Model Checking

Model checking [56], [57] is a framework for verification of finite transition systems using temporal logic [58] as specification formalism. Several formalisms and powerful model checking tools have emerged over the years [59], [60]. The successful application of model checking to various problems [61] could be owed to different factors. Among them, the high automation provided by model checking tools and the possibility to obtain a counterexample when the specification is violated are important ones.

Definition 2: *A finite transition system is a tuple* $\mathcal{T} = \langle S, Act, \rightarrow, I, AP, L \rangle$ *where* S *is a finite set of states,* Act *is a finite set of actions,* $\rightarrow \subseteq S \times Act \times S$ *is a transition relation,* AP *is a finite set of atomic propositions, and* $L : S \rightarrow 2^{AP}$ *is a labelling function.*

Given a transition system $\mathcal{T}$, and a temporal logic formula f, the model checking problem is a decision procedure for $\mathcal{T} \vDash f$. If $\mathcal{T} \nvDash f$, then the model checking algorithm provides a counterexample as evidence for the violation, which can then be used to analyse the issue and the ways to resolve it.

Temporal Logic of Actions

The Temporal Logic of Actions (TLA) is a logical formalism for specifying and reasoning about concurrent systems [62]. TLA is a variant of temporal logic [58] and uses the notion of states and actions to model behavioural properties of systems. TLA, as a logical formalism provides the expressive power to reason about programs using assertions on states and pairs of states (actions). Actions are predicates that relate two consecutive states and are used to capture how the system is allowed to evolve. This section only presents a brief overview of TLA and TLA$^+$, the associated formal language for specifying and model checking systems. A more detailed description of the language and other advanced topics is available in [62]–[64]. The industrial use of TLA$^+$ is documented in [63], which also includes an experience report on using TLA$^+$ for the verification of critical systems at Amazon Web Services [65].

The reasoning system in TLA is built around TLA formulas. A TLA *formula* is true or false on a behaviour. A *behaviour* in TLA is an infinite sequence of states. A *state* in TLA is an assignment of values to variables and a *step* is a pair of states. Steps of a behaviour denote successive pairs of states. Given a system S, with the executions of the system represented as behaviours, and a formula f, S satisfies f if and only if the formula f is true for every behaviour of S.

The elementary building blocks of a TLA formula include state predicates, actions, logical operators (such as $\wedge$,$\neg$, etc.), the temporal operator $\Box$ (always) and the existential quantifier $\exists$. A *state predicate* is a boolean valued expression on states. An action, $\mathcal{A}$, is a boolean valued expression on steps. Actions are formed from unprimed variables and primed variables to represent the relation between old states and new states. The unprimed variables refer to the values of the variables in old states, the first state of the step, whereas the primed variables refer to the variable values in new states, the second state of the step. State predicates have no primed variables. A step is an $\mathcal{A}$-step if it satisfies $\mathcal{A}$. An action is valid, $\vDash \mathcal{A}$, iff every step is an $\mathcal{A}$-step. In TLA, atomic operations of programs are represented by actions.

TLA$^+$ is a formal specification language based on formal set theory, first order logic and TLA. A TLA$^+$ specification, typically denoted *Spec*, is a temporal formula on *behaviours*. All the behaviours satisfying *Spec* constitute the correct behaviours of the system. TLA$^+$ describes a system as a set of behaviours with an initial condition and a next state relation. The initial condition specifies the possible initial states, and the next state relation specifies the possible steps. A TLA$^+$ specification is a temporal formula of the form

$$Spec \triangleq Init \wedge \Box [Next]_{\langle vars \rangle} \wedge Temporal \tag{3.1}$$

where *Init* is a state predicate corresponding to the initial condition, *Next* is an action corresponding to the next state relation, *vars* is a tuple of all variables in the specification, and *Temporal* is a temporal formula usually specifying liveness conditions. Formula *Spec* can be seen as a predicate on behaviours. *Spec* is true for a behaviour σ, if and only if *Init* is true in the first state of σ and every step in σ is either a step that satisfies *Next* or is a *stuttering* step. A *stuttering* step is one in which none of the variables are changed.

The specification (3.1) can be verified using the TLC model checker. TLC takes a TLA$^+$ specification and checks whether the specification satisfies the desired properties by evaluating all possible behaviours of the specification. The TLA$^+$ specification language accompanied by an IDE consisting of TLC and other useful tools can be downloaded from [63].

3.3 Deductive Verification

Model checking is well suited to establish (temporal) properties of state traces, but mostly requires *abstractions* over the real source code. In contrast to that, deductive verification [66] techniques are well suited for fully precise reasoning about the computation on the *source code level*. Often, first order-logic is used to characterise conditions on the data in specific states, in pre- and post-conditions of procedures, or invariants. Deductive verification typically uses a compositional methodology, specifying and verifying one procedure at a time. Verification tools exist for common programming languages such as C [67], Java [68], or Ada [69]. In the following section one such deductive verification tool used in this thesis is presented and in Chapter 4, another deductive verification approach for hybrid systems is presented.

SPARK

Ada [70] is a high level imperative programming language targeting the development of large scale safety critical software. Ada is suited to meet the high integrity software requirements and has been used in several industrial embedded software development projects [71]. SPARK is a subset of Ada with additional features to support formal verification [69]. SPARK uses property specifications in the form of program annotations described inline with the source code to perform static program analysis and build automated proofs to show the correctness of the software. In that sense, SPARK uses the correct-by-construction philosophy through contract-based programming to develop software.

A SPARK program is made up of one or more program units. Subprograms and packages are two examples of SPARK program units. A subprogram execution is invoked by a call and subprograms express a sequence of actions. Procedures and functions are the two types of subprograms in SPARK. Procedure calls are standalone statements, whereas function calls occur in an expression and return a value. Packages group together entities like data types, subprograms, etc., and can be considered to be the equivalent of header files in an object oriented programming language like C++. A program unit consists of two structures, a specification and a body. The specification contains the variables, types and the subprogram declarations with their annotations. The body of a program unit contains the details of the implementation.

Properties are in SPARK specified using subprogram contracts (pre- and post-conditions), loop invariants, and data dependencies. The formal verification toolset in SPARK can perform program analysis on the source code at various levels. Flow analysis capabilities ensure the program correctness with respect to data flow and information flow. Errors arising due to uninitialized variables, data dependencies between inputs and outputs of subprograms, well-formedness of programs, etc., are checked by this level of analysis. A higher level of analysis is to perform automated proofs to check for run time errors and conformance of the program with the specifications. The program annotations specified are used to generate *verification conditions*, which can then be discharged using the proof tools to show program correctness.

CHAPTER 4

Provably Correct Decision-Making

During driving, a human driver takes many critical decisions depending on a lot of factors such as perception, intention, and even emotion. While human decisions and actions may or may not always be logical, decision-making in automated driving systems needs to be logically correct. The main challenge in these systems lies in reliably capturing the different dependent factors in the decision-making process and verifying their correctness.

The decision and control sub-system as shown in Figure 1.1 has different responsibilities that include, (i) long term strategic and short term operational decision-making based on individual and fused sensor inputs, (ii) predicting the motion and intention of the other road users to assess the traffic situation, and (iii) safe path planning for both longer and shorter time horizons. Several approaches can indeed be used to design such sub-systems in autonomous vehicles [11]. More recently, an end-to-end learning approach for autonomous vehicles [72] is becoming popular for complex planning and decision-making processes. However, the verification of such an approach presents unique challenges that pose serious concerns about safety [11], [13], [49], [73].

Autonomous vehicles operate in complex and dynamic environments, which requires decision-making and planning at different levels. For example, to

achieve a goal of autonomously driving from an arbitrary point A to another arbitrary point B, typical high-level strategic decisions could be to decide which lane to drive in, and the associated planning at this level could be to plan a route from A to B [74]. Depending on these high-level decisions, the traffic situation is assessed on the subsequent levels to make low-level decisions such as when to brake, when to change lane, etc. The planning at this level could include planning a path to complete a lane change, or an overtake manoeuvre, etc.

The aim of such decision-making components in these systems is to make safe decisions at all times. In order to make a safe decision, predicting the behaviour of the other road users and assessing the traffic situation is crucial. In addition, the decisions taken by the autonomous vehicle should result in actions that are unambiguous so that the other road users can adapt and co-operate for overall safety. Once the traffic situation is assessed, the decision-making is then based on certain decision thresholds. For example, one way to decide when to brake for a red light is to base the decision on the distance to safely brake without a collision. Similarly, a decision to switch on the turn indicators could depend on whether a lane change process has been initiated or not. In this thesis, we focus on such low-level decision-making where discrete decisions are taken based on certain thresholds that depend on the assessment of the traffic situation and the environment. The following sections describe how formal techniques can be used to prove the correctness of such decision-making algorithms and the insights obtained in the process.

4.1 *Specify* to Prove

As described in Chapter 3, specification is at the heart of the formal verification process. One of the important challenges encountered in Paper A and in Paper C is the lack of adequate documented requirements for the formal verification process. In Paper A, the Lateral State Manager (*LSM*), a decision-making component responsible for managing modes during an autonomous lane change, is formally modelled and one safety requirement *Req.1* (from Paper A):

Req.1 If changing lane, the lane change shall always be to the same side as indicated.

is verified using the different methods. While it proved to be enough to compare different verification approaches and draw insights, it is insufficient to provide a complete safety argument.

Verification of safety requirements at all levels is an important aspect of ISO 26262. A complete safety argument depends on the traceability of the lower level safety requirements to the higher level safety goals. Bergenhem *et al.* [36] claim that the safety requirement refinement process of ISO 26262 implies semantic gaps that present unique challenges to verify completeness and correctness of the safety argument. They also advocate the importance of having formal proofs at each step of the requirement refinement process so that the overall safety argument is complete and correct. Experience from Paper A further strengthens this argument. Though *LSM* was formally verified with respect to *Req.1*, no conclusions can be made towards a complete safety argument.

One way to address the above challenge is to introduce more rigour in the requirement refinement process with the help of formal methods. Based on the insights from Paper A, Paper B presents a model-based safety analysis approach using SCT [51], [75], [76]. The formal approach discussed in Paper B systematically obtains formal models from a given fault tree and analyses them. Since safety analysis techniques like fault tree analysis is used to identify safety goals, i.e., the high-level safety requirements, a model-based approach can potentially be used to achieve traceability in the analysis. For instance, the models used in the safety analysis can incrementally be refined with more details to obtain low-level safety requirements in the subsequent phases.

As an illustration of this approach, consider the *LSM* component studied in Paper A. The *LSM* is a part of the lane change module in the autonomous vehicle. Therefore, in the model-based safety analysis approach, the first task is to create a high-level abstract model of the lane change module. At this level, the interactions and responsibilities of the different sub-components are considered and the implementation details, such as the model of *LSM* as captured in Paper A can be abstracted away. A formal analysis at this level could then be used to obtain the high-level safety requirements. Subsequently, the individual components can be iteratively refined with more details until the implementation model discussed in Paper A is obtained. Paper B only presents how to analyse fault trees and the approach illustrated above is discussed as future work.

Another key insight obtained from the case studies in Paper A and Paper C is that there are unique challenges associated with how a given safety requirement can be formally specified in the different approaches to obtain meaningful verification results. In Paper A, *Req.1* was formally modelled using EFSM as the specification language. The use of additional features such as guards and variables in EFSM definitely made it easier to express *Req.1* with just 2 locations, as shown in Figure 3. In Paper C, an FSM model of *LSM* was automatically learnt. Though theoretically EFSM and FSM have the same expressive power, expressing *Req.1* using FSM was not straightforward.

In the model checking approach using TLA$^+$, *Req.1* was expressed as an invariant property of the model using TLA constructs as shown in Paper A (A.2). In the model checking paradigm, invariant properties are a particular type of the more general safety properties, which are often characterised as *nothing bad should happen* [59]. Invariants, which are linear-time properties in TLA$^+$, are state predicates that should be valid in all reachable states. Several logical constructs are available in TLA$^+$ to express such properties [64]. In comparison, the primary construct in the nonblocking verification algorithms used in the SCT framework is the notion of marked states. Though the constructs in these two formalisms cannot be directly compared, a close translation of the nonblocking verification of marked states would be the branching-time logical[1] construct, **AG EF** *marked states*. **AG EF** *marked states* is a branching-time construct that expresses across **A**ll computational paths **G**lobally, there **E**xists at least one path where **F**inally marked states are reached.

Although similar verification results were obtained from both SCT and model checking, it is well known that logic in branching-time and linear-time are directly incomparable. It could probably be the case that the nature of *LSM* and *Req.1* made it possible to obtain similar results from both these approaches. However, to draw any general conclusions in our context of automated driving, more empirical case studies are required with different types of requirements, especially safety properties that are not necessarily invariants.

With the deductive verification framework in SPARK [69], the major finding in this thesis is that expressing *Req.1* using pre and post-conditions turned out to be hugely inefficient. A primary reason for this is the incompleteness in the requirement refinement mentioned earlier. While *Req.1* in its natural

[1]For an introduction to branching-time logic, refer [59].

language form proved to be sufficient to work with abstract techniques like SCT and model checking, it had to be further broken down in the form of program annotations for meaningful analysis using the deductive framework in SPARK. However, it should not be overlooked that verification of implementation specific safety-critical properties such as division-by-zero, overflow, etc., is automatically done in SPARK with no additional effort.

4.2 Towards Provable Correctness and Completeness

An important take-away message from this thesis is the need for an integrated formal approach to provide completeness in the safety argument. From Paper A and this current chapter of the thesis, we get several insights on the advantages and limitations of the different approaches. At the same time, it is also evident that no method on its own can provide a sufficiently complete safety argument.

The verification techniques in SCT and model checking work on abstract models of the system. Therefore, any correctness proof provided by the method is bound to the model. On the other hand, as seen in Paper A, the deductive framework in SPARK is best suited to verify source code at the implementation level. With such a distinction, experience from Paper A clearly highlights the importance of the verification objective as a factor that affects the choice of the formal method.

The abstract methods like SCT and model checking are indeed best suited to prove correctness of the design in early stages of the development process. However, the use of such methods does not preclude the need for rigour in the implementation stages of the development process. Certainly, there are other ways like strict coding guidelines and standards that can be combined with traditional software testing to assess the quality of the implemented source code. In the end, the question of whether evidence from such arguments meets the rigour provided by a formal proof will be decisive. An ideal approach for the safe deployment of *LSM* would be to introduce the abstract methods in the conceptual design stages and finally use SPARK to have a formally verified implementation of the same.

Another factor that affects the use of formal verification to prove correctness is their ease of integration with the existing software development processes.

With recent advances in continuous software development practices [77], [78], the need for automated development tools that seamlessly integrate into the software pipeline is increased. Such disruptions in the processes have raised concerns for the development of safety-critical systems [79], [80]. In Paper A, it was observed that tools like SUPREMICA and TLA$^+$ are standalone tools with low compatibility with other conventional development tools. However, results from Paper B and Paper C demonstrate that improvements can be made in this regard. The model-based safety analysis approach using SUPREMICA in Paper B is successfully integrated with a systems engineering tool [81] that is widely used in the automotive industry. The formal models automatically learnt using the learning setup in Paper C can also be directly imported into SUPREMICA for subsequent analysis.

4.3 Models - Good, Bad, and Useful

As mentioned in the previous section, the validity of the correctness proof directly corresponds to the validity of the formal model. To answer the question of what would make the model and thereby the proof invalid, two main concerns come to the fore. One, the errors introduced in the modelling process. Two, errors in the assumptions made about the model. Unsurprisingly, these concerns are not specific to the case studies in this thesis but rather a general threat to formal approach based arguments [15], [49].

In Paper A, an existing implementation of the decision-making software was used to manually obtain the formal models in the three different approaches[2]. To overcome the problem of potentially error-prone manual modelling, Paper C presents one possible solution that automatically learns the formal models directly from the implemented code. However, the model validation in Paper A and Paper C was primarily done by simulation and manual reviews. While the approach in Paper C is promising, more such empirical case studies need to be done to achieve sufficient confidence in scaling the approach.

A significant part of the manual effort needed in SPARK is in obtaining the program annotations that accurately formalize the safety requirements. A fundamental trade-off exists between the level of proof automation and the

[2]In the case of SPARK, the code was re-implemented using Ada to make use of the formal verification framework.

expressiveness of the underlying logic in the deductive framework. Therefore, a correctness proof for complex safety requirements would require higher logical expressiveness, thereby making manual effort inevitable.

Formal models of the *LSM* were obtained manually in Paper A and automatically in Paper C. In both these processes, certain assumptions and abstractions of the model had to be made. For instance, to formally reason about the *LSM*, only the decision logic was modelled and its interactions with the other components of the lane change module such as the strategic planner (Figure 1 in Paper A) were assumed to be correct. Such interactions and external dependencies of the *LSM* were abstracted and modelled as discrete variables that affect the state transitions of the model.

As discussed in Paper A and Paper C, these assumptions and abstractions had to be made to deal with the well known state-space explosion problem associated with approaches like SCT and model checking. In some cases, such an abstraction was forced due to the limitations in expressiveness of the underlying formalism. For example, the decision on whether to change lane or not is dependent on the assessment of whether there is sufficient time to safely (collision-free) move to the new lane. Such an assessment involves state variables that vary continuously with time. Clearly, this notion of continuous state change requires suitable abstractions in the modelling formalisms of the SCT and model checking approaches discussed in this thesis. Since formal verification in SPARK is applied at the source-code level, we only have to deal with arithmetic manipulations of the supported data-types. However, the challenge is manifested in the form of extensive program annotations and the complex proof effort required to even prove simple functional safety requirements.

Such modelling inaccuracies also highlight the implicit trade-off between false alarms and vacuous truth in these methods. While abstracting too many details increases the chance of potential false alarms, having a model with finer details and restrictive constraints could potentially end up with a proof that is vacuously true. Since it is quite evident that assessing the situation is crucial for the correctness of decision-making in AD, it is definitely beneficial to resort to a logical formalism that can capture both discrete state changes as well as the continuous dynamics of the system. Having such a formalism allows us to reason about a model that is closer to reality so that the risk of having invalid proofs due to invalid assumptions is reduced. This brings us to the field of hybrid system verification that is briefly described in the next section.

4.4 Hybrid System Verification

By hybrid systems, we mean mathematical models of systems that combine discrete dynamics (behaviour that changes discretely) with continuous dynamics (behaviour that changes continuously with time). A logical formalism that supports the specification and verification of hybrid systems is *differential dynamic logic* (dL) [46], [47]. KeYmaera X [82] is a theorem prover that implements dL and its associated verification techniques. dL has been used to model and prove properties of hybrid systems in automotive [32], [83], [84] and autonomous robotics [85].

To model hybrid systems, dL has the notation of *hybrid programs* (HP) [46], [47]. Hybrid programs consist of different program statements including the use of differential equations to describe continuous behaviour. A summary of the program statements in HP, taken from [86], is shown in Table 4.1.

Statement	Effect
$\alpha;\beta$	sequential composition where β starts after α finishes
$\alpha \cup \beta$	nondeterministic choice, follow either α or β
α^*	nondeterministic repetition, repeat α n times for any $n \in \mathbb{N}_0$
$x := \theta$	discrete assignment of the value of term θ to variable x
$x := *$	nondeterministic assignment of an arbitrary real number to x
$?F$	test if formula F holds in the current state, abort otherwise
$(x_1' = \theta_1, \ldots,$	continuous evolution of x_i along the differential equation system
$x_n' = \theta_n \& F)$	$x_i' = \theta$ restricted to evolution domain F

Table 4.1: Statements of hybrid programs from [86]. F is a first-order formula, α, β are hybrid programs.

The nondeterministic operators help address two critical aspects in proving correctness of AD: (i) they can be used to describe unknown behaviour, which is typically the case in modelling the highly unpredictable environment for AD systems; (ii) nondeterminism helps achieve a maximally permissive proof of correctness by reducing the dependency of the proof on the assumptions in the model. For example, to prove the correctness of a decision-making algorithm, its interactions with the other components (e.g. sensor fusion, planning, etc.) in the AD system can be modelled with nondeterministic behaviour. In such a case, a correctness proof of the decision-making algorithm is free from assumptions on the component interactions and would still be valid if changes were later introduced in the implementation of the other

components.

The formulas of dL include formulas of first-order logic of real arithmetic and the modal operators $[\alpha]$ and $\langle\alpha\rangle$ for any HP α [46], [47]. The set of formulas generated by dL is given by the following grammar

$$P, Q ::= \theta_1 \sim \theta_2 \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \rightarrow Q \mid P \leftrightarrow Q \mid \forall x P \mid \exists x P \mid [\alpha]P \mid \langle\alpha\rangle P$$

where P, Q are dL formulas, θ_1, θ_2 are polynomial terms (arithmetic expressions) over reals, $\sim$ is any operator in $\{<, \leq, =, \neq, \geq, >\}$, x is a variable, and α is an HP. The dL formula $\langle\alpha\rangle P$ expresses that there is at-least one non-aborting run of α that leads to a state in which the dL formula P is true. The dL formula $[\alpha]P$ expresses that every non-aborting run of HP α leads to a state in which the the dL formula P is true. Similarly, to specify the correctness of the HP α, we use a dL formula of the form $P \rightarrow [\alpha]\, Q$, which states that if started at an initial state where formula P is true, then all (non-aborting) runs of α only lead to states where formula Q is true. In our context of AD, this can easily be translated to a dL formula that reflects the *assume-guarantee* type of requirement:

$$(assume) \rightarrow [\ \texttt{HP}\ ]\ (guarantee) \tag{4.1}$$

where HP is the hybrid program describing the system, which includes discrete and continuous dynamics. HP can further be refined using the sequential operator (;) in dL as:

$$[\ \texttt{HP}\ ] \ \triangleq\ [(env; ctrl; plant)^*] \tag{4.2}$$

where *env* models the behaviour of the environment, while *ctrl*, and *plant* model the discrete and continuous behaviour, respectively. In the following sections, an illustration of how dL can be used to model a decision and control component of an AD system is presented. Proving the correctness of the component is ongoing work and the following sections give a brief discussion of the current research direction.

The Safety Monitor

The illustrative example used in this section is the *safety monitor*, which is a part of the decision and control sub-system in an autonomous vehicle.

During normal driving conditions, the autonomous vehicle (hereafter referred as the *ego-vehicle*) is subjected to different types of constraints, for instance constraints to ensure safe vehicle motion, constraints for smooth and efficient driving, etc. The safety constraints are crucial to keep a safe speed with respect to the road and other road users. The safety constraints restrict the speed of the ego-vehicle to deal with threats in the surrounding traffic. The constraints are set by a pair of values, a critical speed, `vCrit`, and a critical position, `xCrit`. The critical speed `vCrit` is the estimated maximum speed that the ego-vehicle can have at position `xCrit` in order to guarantee safety. The objective of the safety monitor is to calculate the required acceleration to fulfil the safety constraint at all times.

A simplified overview of the system architecture is presented in Fig. 4.1. Depending on the traffic state and the vehicle state information received from the sense sub-system, the constraint generation component calculates and communicates the constraint to the threat assessment component of the safety monitor. Based on the current vehicle state and the constraint, the safety monitor performs a threat assessment and calculates an acceleration value, `aSafe` which is the acceleration required to fulfil the safety constraint. In addition, the decision and control subsystem also includes a nominal control component which calculates a nominal acceleration request `aNom`. This acceleration request from the nominal control block is responsible for the ego-vehicle operation under normal driving conditions. The nominal control is a representation of any feature in the ego-vehicle (for instance comfort control, cruise control, etc.) and therefore at any time it can request to accelerate or decelerate at will depending on the respective objective.

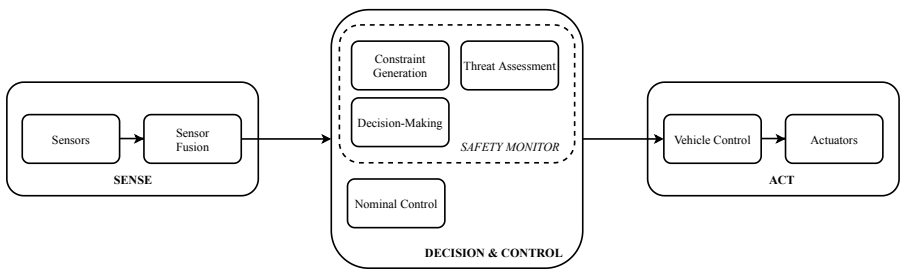


Figure 4.1: System architecture of the safety monitor

The objective of the vehicle control component is to determine a control policy to solve the set-point tracking problem where it tracks the acceleration request received. The interactions between the different components in the decision and control module and with vehicle control is intentionally not shown in Figure 4.1 as one of the research questions here is whether dL and KeYmaera X could be used to identify a safe system architecture. For instance, whether the safety monitor needs to know aNom for safe decision-making, and under what conditions should vehicle control follow aSafe and aNom, are important questions that need to be answered.

In summary, nominal control is responsible for normal driving conditions and if the ego-vehicle is about to violate the safety constraint, the safety monitor ensures safety by issuing brake commands through aSafe. Since the safety monitor is responsible for safe and emergency braking, it has higher braking capabilities than the nominal control.

From the description, it is obvious that in order to reason about the correctness of the safety monitor, we need to consider the continuous dynamics as well as the discrete control decisions in our model. The primary objective of the formal design process is to provide a strong formal argument that the control decisions will always be safe, i.e., the ego-vehicle at position x and velocity v always respects the safety constraint, and the ego-vehicle speed v never exceeds vCrit at the critical position xCrit. Therefore, the safety guarantee to prove is given by,

$$\textit{guarantee} \equiv (\mathtt{x} \geq \mathtt{xCrit} \rightarrow \mathtt{v} \leq \mathtt{vCrit}). \tag{4.3}$$

Ego-vehicle Motion Model

While it is beneficial to have a model that is close to reality, a complex model increases the proof complexity and traceability. An efficient workflow is to start simple and develop increasingly complex models until the desired model fidelity is achieved. In that vein, we first start with a constant acceleration model, i.e., the position x of the ego-vehicle is given by $\mathtt{x}'' = \mathtt{a}$ to capture the continuous dynamics. In this model, the acceleration values are applied instantaneously and remain constant between subsequent control decisions. In addition, since we focus on the safety monitor, we do not model the algorithms in the vehicle control and instead assume a control policy that faithfully follows the acceleration requests. For instance, an acceleration request of $\mathtt{a} = 5\,\mathrm{m/s^2}$

is instantaneously applied and its effect is observed in the continuous evolution of the differential system. With these considerations, the ego-vehicle motion model is:

$$motion \equiv t := 0; \{x' = v,\ v' = a,\ t' = 1\ \&\ v \geq 0 \ \wedge\ t \leq T\}. \tag{4.4}$$

Based on the constant acceleration model, the time derivative of the ego-vehicle's position `x` is its velocity `v` and the time derivative of `v` is the acceleration `a`, as given by the differential equations $x' = v$ and $v' = a$ respectively. The sample time of the system is modelled with a timer t that is reset to $t := 0$ before the differential system evolves. The evolution of the timer is given by the differential equation $t' = 1$ and is bound by the *evolution domain constraint* $t \leq T$. Evolution domain constraints in differential equations introduce bounds on the continuous dynamics and restricts the continuous evolution of the system to stay within that domain [47]. Thus the system follows differential equation $t' = 1$ for any duration while inside the region $t \leq T$, but is never allowed to leave that region. Therefore the system evolution stops before $t \leq T$ evaluates to false, for the discrete control to make appropriate decisions. Intuitively, this captures the notion that the system will ensure that a discrete control decision is made after at most T time units. Similarly, another evolution domain $v \geq 0$ is included to capture the notion that braking with negative acceleration value does not make the ego-vehicle drive backwards.

Constraint Generation

The constraint generation module calculates and updates the constraints `xCrit` and `vCrit`. One possible approach here is to model the algorithm used as a hybrid program and subsequently prove the safety guarantee (4.3). However, such an approach will only provide the safety argument for one such implementation of the constraint generation. In order to have a maximally permissive design and a safety argument that covers a wide variety of constraint generation algorithms, we make use of the nondeterministic assignment as shown in (4.5). In the model, it also necessary to ensure that the constraint generation does not pick arbitrary values that are impossible to satisfy like negative critical speed and constraints that are beyond the physical capabilities of the

ego-vehicle.

$$cg \equiv \texttt{xCrit} := *; \texttt{vCrit} := *; ?(\texttt{vCrit} \geq 0 \wedge \texttt{limit}) \tag{4.5}$$

$$\texttt{limit} \equiv (\texttt{v}^2 - \texttt{vCrit}^2) \leq 2\,\texttt{aMinS}\,(\texttt{xCrit-x}) \tag{4.6}$$

The test condition in (4.5) ensures that the assigned arbitrary constraints satisfy the physics of the vehicle. The condition `limit` (4.6) characterizes the relationship between the ego-vehicle's current speed `v`, position `x`, maximum braking capability of the safety monitor `aMinS`, and the safety constraint. This condition can be calculated by solving the differential equations of the motion model in (4.4):

$$\{x' = v,\ v' = a\}.$$

Integrating the above differential equations over time, we get the following set of equations:

$$x = x_0 + v_0 t + \frac{at^2}{2} \tag{4.7}$$

$$v = v_0 + at \tag{4.8}$$

where x_0 and v_0 are the initial position and initial velocity respectively. From (4.8), we can see that using the maximum braking capability $-aMinS$, the time required to reach the critical velocity `vCrit` from current velocity `v` is given by

$$t = \frac{v - vCrit}{aMinS}. \tag{4.9}$$

Since (4.7) gives the solution to the distance travelled, we can calculate the braking distance travelled while slowing down from current velocity v to critical velocity vCrit as the minimum distance for the critical position xCrit

$$xCrit \geq x + v\left(\frac{v - vCrit}{aMinS}\right) - \frac{aMinS}{2}\left(\frac{v - vCrit}{aMinS}\right)^2 \tag{4.10}$$

It is straightforward to simplify (4.10) to obtain the equivalent condition `limit` in (4.6).

Finally, to put everything together so that it corresponds to the (assume-

guarantee) formula in (4.1), we get the following refinement:

$$\begin{aligned} &(assume) \rightarrow [\ \texttt{HP}\ ]\ (guarantee) \\ \equiv\ &(assume) \rightarrow [(env; ctrl; plant)^*]\ (guarantee) \\ \equiv\ &(assume) \rightarrow [(cg; ctrl; motion)^*]\ (\texttt{x} \geq \texttt{xCrit} \rightarrow \texttt{v} \leq \texttt{vCrit}) \end{aligned}$$

The challenge in obtaining a safety proof for the safety monitor is to identify the conditions for safe decision-making in *ctrl* and how the interactions of the safety monitor with the other components affect those decisions. dL and KeYmaera X provides the necessary logical formalism and verification techniques to achieve this, and this is one of the immediate future work direction.

CHAPTER 5

Summary of Included Papers

Paper A In Paper A, three different formal verification approaches, namely supervisory control theory, model checking, and deductive verification are used to formally verify an existing decision-making software in an autonomous vehicle. The three approaches are evaluated to identify (i) the challenges in applying formal verification to AD and (ii) the factors that affect the choice of the verification method. Insights from Paper A show the need for an integrated formal approach to prove correctness. Paper A discusses how the verification objective differs in the three approaches and presents the challenges in formally modelling and specifying the decision-making software.

Paper B One of the challenges identified in Paper A is the lack of adequate documented requirements for the formal verification process. Safety analysis methods like Fault Tree Analysis is widely used to identify high-level safety requirements. By introducing more rigour using formal methods, challenges in the requirement refinement process can be addressed. In this regard, Paper B presents a model-based safety analysis approach using Supervisory Control Theory. In Paper B, a systematic approach to incrementally obtain formal models from a fault tree is pre-

sented. Paper B also shows how the formal models can be analysed in SUPREMICA and also how important quality factors like minimal cut sets can be calculated in the presented approach.

Paper C Another challenge identified in Paper A is the manual effort required in obtaining formal models. Manual construction of formal models is expensive, error-prone, and intractable for large systems. As one possible solution to this problem, Paper C applies active automata learning techniques to obtain formal models of the decision-making software studied in Paper A. Results from Paper C demonstrate the feasibility of such automated techniques for automotive industrial use. Two such learning algorithms are evaluated and practical challenges in their application are presented. Furthermore, insights from Paper C show that such techniques could potentially pave way for the widespread adoption of formal methods to guarantee software correctness of AD systems.

CHAPTER 6

Concluding Remarks and Future Work

This thesis started with the hypothesis that formal methods, especially formal verification, can be used to prove the correctness of AD systems and thereby aid in their safe deployment. Results from this thesis provide strong evidence to support that argument. To achieve the overall goal of establishing formal verification as an efficient tool in AD software development, three research questions are considered. The first research question, **RQ 1** is to identify the factors that affect the application of formal verification and the current challenges in existing methods. Based on the results and insights from the included papers, some important challenges are identified and discussed. A key obstacle to the application of formal verification is the lack of adequate requirements at all levels of the development process. This thesis identifies the need for the introduction of more rigour in the requirement refinement process and presents one possible solution by using a formal model-based safety analysis approach.

Another key finding is the need for an integrated formal approach to prove correctness and to provide a complete safety argument. Three different formal approaches are evaluated and a discussion on how the verification objective differs in each of them is provided. Finally, two principal concerns associated

with the formal modelling process are identified and discussed; one: modelling inaccuracies due to manual errors; and two: the validity of the assumptions and the abstractions made in the formal model. Such inaccuracies could potentially result in an invalid correctness proof and viable options to combat them are presented.

The second research question, **RQ 2** concerns how to address the identified challenges. One way to address the problems associated with manual modelling, is to automatically learn the formal models directly from the code. Similarly, to address the gaps in the requirement refinement process, the model-based safety analysis approach discussed in Paper B can be used to systematically analyse safety models and incrementally obtain safety requirements at different levels as discussed in Chapter 4. Demonstrations from Paper B and Paper C show that these solutions can successfully be integrated with day-to-day software development activities, thereby providing valuable insights to answer **RQ 3**.

6.1 Future Work

Based on the insights obtained, it is quite evident that in order to reason about the correctness of decision-making in AD, it is crucial to also reason about the correctness of the algorithms assessing the traffic situation and environment. Therefore, proving correctness using a formalism that supports hybrid systems such as the differential dynamic logic discussed in Section 4.4 is necessary. One possible future research direction is to investigate whether such an approach enables us to obtain a correctness proof that provides even a higher level of safety assurance than the methods discussed in Paper A. The other observation made is the need for many more empirical studies to further strengthen the conclusions made and possibly also obtain new ones regarding provable correctness of AD. Any future work in this direction is most definitely rewarding.

References

[1] World Health Organization (WHO), *Road Traffic Injuries*, https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries, Accessed: 2020-08-02, 2020.

[2] National Highway Traffic Safety Administration (NHTSA), *Automated Vehicles for Safety*, https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety, Accessed: 2020-08-02.

[3] T. Litman, *Autonomous vehicle implementation predictions*. Victoria, Canada: Victoria Transport Policy Institute, 2020.

[4] SAE On-Road Automated Vehicle Standards Committee and others, "Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles", *SAE International: Warrendale, PA, USA*, 2018.

[5] Society of Automotive Engineers (SAE) International, *SAE Standards: J3016 automated-driving graphic update*, https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic, Accessed: 2020-08-02, 2019.

[6] The Insurance Institute for Highway Safety, *New studies highlight driver confusion about automated systems*, https://www.iihs.org/news/detail/new-studies-highlight-driver-confusion-about-automated-systems, Accessed: 2020-08-02, 2019.

[7] MIT Technology Review, *The three challenges keeping cars from being fully autonomous*, `https://www.technologyreview.com/2019/04/23/103181/the-three-challenges-keeping-cars-from-being-fully-autonomous/`, Accessed: 2020-08-02, 2019.

[8] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles", *IEEE Transactions on intelligent vehicles*, vol. 1, no. 1, pp. 33–55, 2016.

[9] A. Mohamed, J. Ren, M. El-Gindy, H. Lang, and A. Ouda, "Literature survey for autonomous vehicles: Sensor fusion, computer vision, system identification and fault tolerance", *International Journal of Automation and Control*, vol. 12, no. 4, pp. 555–581, 2018.

[10] Z. Wang, Y. Wu, and Q. Niu, "Multi-sensor fusion in automated driving: A survey", *IEEE Access*, vol. 8, pp. 2847–2868, 2019.

[11] W. Schwarting, J. Alonso-Mora, and D. Rus, "Planning and decision-making for autonomous vehicles", *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[12] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?", *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[13] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation", *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[14] Koopman, Philip and Wagner, Michael, "Autonomous vehicle safety: An interdisciplinary challenge", *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, 2017.

[15] P. Koopman, A. Kane, and J. Black, "Credible autonomy safety argumentation", in *27th Safety-Critical Systems Symposium*, 2019.

[16] S. Riedmaier, T. Ponn, D. Ludwig, B. Schick, and F. Diermeyer, "Survey on scenario-based safety assessment of automated vehicles", *IEEE Access*, vol. 8, pp. 87 456–87 477, 2020.

[17] The Insurance Institute for Highway Safety, *Fatality Facts 2018*, `https://www.iihs.org/topics/fatality-statistics/detail/state-by-state`, Accessed: 2020-08-02, 2018.

[18] Philip Koopman, *Potentially deadly automotive software defects*, https://betterembsw.blogspot.com/2018/09/potentially-deadly-automotive-software.html, Accessed: 2020-08-02, 2018.

[19] Original Equipment Suppliers Association, *Automotive Defect and Recall Report*, https://www.oesa.org/sites/default/files/automotive-defect-recall-report-2019.pdf, Accessed: 2020-08-02, 2019.

[20] Global News, *Tesla's Autopilot system faces increased scrutiny after 3 crashes, 3 deaths*, https://globalnews.ca/news/6363342/tesla-autopilot-crashes-deaths/, Accessed: 2020-08-02, 2020.

[21] The NewYork Times, *Tesla Autopilot System Found Probably at Fault in 2018 Crash*, https://www.nytimes.com/2020/02/25/business/tesla-autopilot-ntsb.html, Accessed: 2020-08-02, 2020.

[22] Business Insider, *Uber self-driving cars were involved in 37 crashes before a fatal incident*, https://www.businessinsider.com/uber-test-vehicles-involved-in-37-crashes-before-fatal-self-driving-incident-2019-11?r=US&IR=T, Accessed: 2020-08-02, 2019.

[23] M. Blanco, J. Atwood, S. M. Russell, T. Trimble, J. A. McClafferty, and M. A. Perez, "Automated vehicle crash rate comparison using naturalistic data", Virginia Tech Transportation Institute, Tech. Rep., 2016.

[24] WIRED, *Google's Self-Driving Car Caused Its First Crash*, https://www.wired.com/2016/02/googles-self-driving-car-may-caused-first-crash/, Accessed: 2020-08-02, 2016.

[25] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal verification of avionics software products", in *International symposium on formal methods*, Springer, 2009, pp. 532–546.

[26] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano, "A formal verification environment for railway signaling system design", *Formal Methods in System Design*, vol. 12, no. 2, pp. 139–161, 1998.

[27] M. Lawford and A. Wassyng, "Formal verification of nuclear systems: Past, present, and future", in *1st International Workshop on Critical Infrastructure Safety and Security (CrISS-DESSERT'11)*, vol. 1, 2011, pp. 43–51.

[28] G. Bahig and A. El-Kadi, "Formal verification of automotive design in compliance with iso 26262 design verification guidelines", *IEEE Access*, vol. 5, pp. 4505–4516, 2017.

[29] V. Todorov, F. Boulanger, and S. Taha, "Formal verification of automotive embedded software", in *Proceedings of the 6th Conference on Formal Methods in Software Engineering*, 2018, pp. 84–87.

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

[31] D. Gurov, C. Lidström, M. Nyberg, and J. Westman, "Deductive functional verification of safety-critical embedded c-code: An experience report", in *Critical Systems: Formal Methods and Automated Verification*, Springer, 2017, pp. 3–18.

[32] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified", in *International Symposium on Formal Methods*, Springer, 2011, pp. 42–56.

[33] RTCA (Firm). SC 167, *Software considerations in airborne systems and equipment certification*. RTCA, Incorporated, 2011.

[34] International Electrotechnical Commission, "IEC 62279: Railway applications—Communications, signalling and processing systems—Software for railway control and protection systems", *International Electrotechnical Commission: Geneva, Switzerland*, 2002.

[35] ISO, "ISO 26262:2018–Road vehicles–Functional safety", *International Standard ISO/FDIS*, 2018.

[36] 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 *CARS 2015 – Critical Automotive applications: Robustness & Safety, Paris, France*, 2015.

[37] N. B. Ruparelia, "Software development lifecycle models", *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[38] ANSI/UL, *ANSI/UL 4600 - Standard for Evaluation of Autonomous Products*, `https://ul.org/UL4600`, 2020.

[39] ISO, "ISO/PAS 21448:2019–Road vehicles–Safety of the intended functionality", *International Standard ISO/FDIS*, 2019.

[40] S. Agerholm and P. G. Larsen, "A lightweight approach to formal methods", in *International Workshop on Current Trends in Applied Formal Methods*, Springer, 1998, pp. 168–183.

[41] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. De Sousa, *Rigorous software development: an introduction to program verification*. Springer Science & Business Media, 2011.

[42] G. J. Holzmann, "The model checker spin", *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[43] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, "Deductive software verification–the key book", *Lecture notes in computer science*, vol. 10001, 2016.

[44] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems", in *International conference on computer aided verification*, Springer, 2011, pp. 585–591.

[45] M. Althoff, "Cora 2016 manual", *Technische Universitat Munchen, Garching, Germany*, 2016.

[46] A. Platzer, "Differential dynamic logic for hybrid systems", *Journal of Automated Reasoning*, vol. 41, no. 2, pp. 143–189, 2008.

[47] Platzer, André, *Logical foundations of cyber-physical systems*. Springer, 2018, vol. 662.

[48] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey", *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, 2017.

[49] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey", *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–41, 2019.

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

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

[52] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional nonblocking verification of extended finite-state machines", *Discrete Event Dynamic Systems*, vol. 26, no. 1, pp. 33–84, 2016.

[53] J. Krook, A. Zita, R. Kianfar, S. Mohajerani, and M. Fabian, "Modeling and synthesis of the lane change function of an autonomous vehicle", *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 133–138, 2018.

[54] M. Skoldstam, K. Akesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables", in *2007 46th IEEE Conference on Decision and Control*, IEEE, 2007, pp. 3387–3392.

[55] R. Malik, "Programming a fast explicit conflict checker", in *2016 13th International Workshop on Discrete Event Systems (WODES)*, IEEE, 2016, pp. 438–443.

[56] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic", in *Workshop on Logic of Programs*, Springer, 1981, pp. 52–71.

[57] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR", in *International Symposium on programming*, Springer, 1982, pp. 337–351.

[58] A. Pnueli, "The temporal logic of programs", in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, IEEE, 1977, pp. 46–57.

[59] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[60] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018.

[61] E. M. Clarke and Q. Wang, "2^5 Years of Model Checking", in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, Springer, 2014, pp. 26–40.

[62] L. Lamport, "The temporal logic of actions", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.

[63] L. Lamport, *The TLA$^+$ home page*, `https://lamport.azurewebsites.net/tla/tla.html`, Accessed: 2019-04-22.

[64] L. Lamport, *Specifying systems: the TLA$^+$ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[65] C. Newcombe, "Why amazon chose TLA$^+$", in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Springer, 2014, pp. 25–39.

[66] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 583, Oct. 1969.

[67] N. Kosmatov, V. Prevosto, and J. Signoles, "A lesson on proof of programs with Frama-C. Invited tutorial paper", in *Tests and Proofs*, M. Veanes and L. Viganò, Eds., Springer, 2013, ISBN: 978-3-642-38916-0.

[68] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification—The KeY Book*, ser. LNCS. Springer, 2016, vol. 10001.

[69] J. Barnes, *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, 2012.

[70] Barnes, John, *Programming in Ada 2012*. Cambridge University Press, 2014.

[71] *Adacore - homepage*, `https://www.adacore.com/`, Accessed: 2019-04-26.

[72] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars", *arXiv preprint arXiv:1604.07316*, 2016.

[73] F. Al-Khoury, *Safety of machine learning systems in autonomous driving*, 2017.

[74] J. Krook, L. Svensson, Y. Li, L. Feng, and M. Fabian, "Design and formal verification of a safe stop supervisor for an automated vehicle", in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 5607–5613.

[75] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes", *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[76] W. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems: A brief history", *Annual Reviews in Control*, vol. 45, pp. 250–256, 2018.

[77] J. Bosch, "Continuous software engineering: An introduction", in *Continuous software engineering*, Springer, 2014, pp. 3–13.

[78] S. Vöst and S. Wagner, "Towards continuous integration and continuous delivery in the automotive industry", *arXiv preprint arXiv:1612.04139*, 2016.

[79] S. Vost and S. Wagner, "Keeping continuous deliveries safe", in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, pp. 259–261.

[80] R. Kasauli, E. Knauss, B. Kanagwa, A. Nilsson, and G. Calikli, "Safety-critical systems and agile development: A mapping study", in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2018.

[81] SYSTEMITE, *Systemweaver*, `https://www.systemweaver.se/`, Accessed: 2020-05-09.

[82] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "Keymaera x: An axiomatic tactical theorem prover for hybrid systems", in *International Conference on Automated Deduction*, Springer, 2015, pp. 527–538.

[83] S. Mitsch, S. M. Loos, and A. Platzer, "Towards formal verification of freeway traffic control", in *2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*, IEEE, 2012, pp. 171–180.

[84] S. M. Loos and A. Platzer, "Safe intersections: At the crossing of hybrid systems and verification", in *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, IEEE, 2011, pp. 1181–1186.

[85] S. Mitsch, K. Ghorbal, and A. Platzer, "On provably safe obstacle avoidance for autonomous robotic ground vehicles", 2013.

[86] J.-D. Quesel, S. Mitsch, S. Loos, N. Aréchiga, and A. Platzer, "How to model and prove hybrid systems with keymaera: A tutorial on safety", *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 1, pp. 67–91, 2016.

Part II

Papers

PAPER A

Verification of Decision Making Software in an Autonomous Vehicle: An Industrial Case Study

Yuvaraj Selvaraj, Wolfgang Ahrendt, Martin Fabian

Larsen K., Willemse T. (eds) Formal Methods for Industrial Critical Systems (FMICS 2019), Lecture Notes in Computer Science,
vol 11687, pp. 143–159, Aug. 2019.
 DOI: 10.1007/978-3-030-27008-7_9

The layout has been revised.

Abstract

Correctness of autonomous driving systems is crucial as incorrect behaviour may have catastrophic consequences. Many different hardware and software components (e.g. sensing, decision making, actuation, and control) interact to solve the autonomous driving task, leading to a level of complexity that brings new challenges for the formal verification community. Though formal verification has been used to prove correctness of software, there are significant challenges in transferring such techniques to an agile software development process and to ensure widespread industrial adoption. In the light of these challenges, the identification of appropriate formalisms, and consequently the right verification tools, has significant impact on addressing them. In this paper, we evaluate the application of different formal techniques from supervisory control theory, model checking, and deductive verification to verify existing decision and control software (in development) for an autonomous vehicle. We discuss how the verification objective differs with respect to the choice of formalism and the level of formality that can be applied. Insights from the case study show a need for multiple formal methods to prove correctness, the difficulty to capture the right level of abstraction to model and specify the formal properties for the verification objectives.

1 Introduction and Related Work

Significant progress has lately been made in the global automotive industry towards autonomous vehicles. Autonomous vehicles can potentially increase road safety and help reduce road traffic accidents. However, these are extremely complex safety critical systems, and human safety depends on their correctness. The level of complexity in these systems is manually intractable. Factors like size, structure (level of interaction and communication between different systems), environment (the physical world in the case of autonomous vehicles), application domain etc., all contribute to the complexity. It is im-

perative that all safety critical parts of an autonomous vehicle are veritably reliable and safe. This is a challenge for the development process due to the complexity needed to be managed not only in the design but also in the verification and validation process.

An autonomous vehicle consists of many software and hardware components interacting to solve different tasks, ranging from sensing, decision making, and planning to actuation and control. The level of complexity involved may lead to subtle but potentially dangerous bugs arising due to unforeseen edge cases, errors in the software design and/or implementation. Coverage based testing is a widely adopted work flow in many large scale software development companies, but exhaustive testing is not tractable. Testing can never guarantee absence of unintended consequences nor provide sufficient certification evidence in all cases. Thus, there is a need for complementary methods to guarantee system safety, and the use of formal methods for this is becoming prevalent [1], [2].

The international standard ISO 26262 [3] provides guidance on a risk based approach to manage, specify, develop, integrate, and verify safety critical systems in road vehicles, including various references to formal specification and verification. Adherence to the standard can potentially ensure that system quality is maintained, and unreasonable residual risk is avoided. The standard is based upon the V model of product development [4] and aims at achieving system safety through safety measures implemented at various levels of the development process. However, the standard addresses neither specific challenges inherent to autonomous driving systems, nor the development of safety critical software in an agile development work flow.

Thus, research is needed to solve challenges arising from such inter-disciplinary problems, and these challenges are at-least two fold:

1. The application of formal verification to autonomous driving systems;

2. The transfer of formal verification techniques to large scale agile development of safety critical software.

The first challenge is relatively new and is driven by recent developments in autonomous systems. The second challenge relates to a long standing problem of successful industrial adoption of formal techniques in software development. However, the addition of agile methods to safety critical software development has introduced new directions.

Formal methods—with varying levels of formalisation—can be applied at various stages of the software development process. The choice of verification method and the expressive power of the formalism used to specify the properties is an important choice that affects the conclusions drawn from the results of the verification process. In this paper, we evaluate three formal verification methods and their respective formalisms to verify existing software in an autonomous driving vehicle: Supervisory Control Theory with Extended Finite State Machines [5], [6], Model Checking with Temporal Logic of Actions [7], Deductive Verification with contract based programming [8]. We discuss how the verification objective differs in these methods and how multiple formal methods can help tackle the challenges in industrial autonomous driving software development.

A recent survey [2] on formal specification and verification of autonomous robotic systems is a comprehensive study of current state-of-the art literature focused on formal modelling, formal specification, and formal verification of robotic systems. It gives a summary on the challenges faced, current methods in tackling the challenges, and the limitations of existing methods. In [9], an overview of the challenges in designing, specifying and verifying cyber-physical systems, particularly semi-autonomous driving systems with human interaction is provided. [10] presents a model checking framework for verifying autonomous systems with a distinguished rational 'agent', confined to the system architecture level with autonomous driving as one example scenario. There are prior research focused on the development of autonomous systems in a generic sense [1], [2], surveys on tool based verification methods and tools [11], [12], and the general industrial adoption of formal methods technology [13]–[16].

In contrast to the literature cited above, our work is specific to autonomous driving and we discuss a tightly coupled approach to tackle the two-fold challenge with an industrial case study. The problem description is given in Section 2, followed by separate sections for the three different verification approaches handled in this paper. Section 6 discusses the evaluation and insights from the industrial case study. The paper concludes with some remarks in Section 7.

2 Problem Description

Zenuity is one of the leading companies in the development of safe and reliable autonomous driving software. A significant part of the embedded software developed at Zenuity is safety critical. In [17], formal verification was applied to a small part of the autonomous driving software in development and non-conformance to a few basic specifications was reported. The work presented in this paper is a continuation of the work started in [17].

The focus of this paper is a sub-module of the decision making and planning module, called *Lateral State Manager* (*LSM*), which solves the sub-function of managing modes during a lane change. A simplified overview of the system and the interactions are shown in Fig. 1. The software module is implemented in object-oriented MATLAB-code using several classes, each solving different sub-problems. The interaction of the *LSM* class with a high level strategic planner (*Planner*) and a low level planner (*Path Planner*) is also shown in Fig. 1.

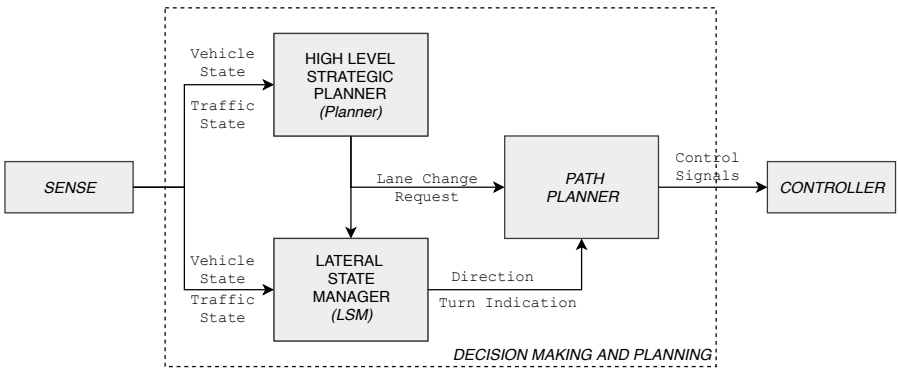


Figure 1: System overview and interactions.

The *Planner* in the lane change module is responsible for strategic decisions and depending on the state of the vehicle, the *Planner* sends lane change requests to the *LSM*, indicating the desired lane to drive in. These requests are in the form of `NoRequest`, `ChangeLeft`, and `ChangeRight`. On receiving a request, the *LSM* keeps track of the lane change process by managing the different modes possible during the process, and issues commands to the *Path Planner*. If a lane change is requested, the *Path Planner* sends control signals

to the low level controller to perform a safe and efficient lane change. Due to the inherent nature of the task to solve, the *LSM* implements a finite state machine. An example of a state in the *LSM* state machine is *State_Finished* that represents the completion of the lane change process.

A call to *LSM* is issued at every execution cycle. During each call, the *LSM* undergoes three distinct execution stages. First, all the inputs are updated according to the function call arguments. Second, depending on the current state, code is executed to decide whether the system transits to a new state or not. This code also assigns outputs and persistent variables. Finally, if a transition is performed, the last stage executes code corresponding to the new state entered and assigns new values to the variables.

Of course, *LSM* is safety critical and its correctness is crucial. In our work, we focus on verifying properties that affect the safety of the system, i.e. a violation of which will result in an unsafe behaviour. From a software development perspective, these properties are typically stated as safety requirements. In [17], one such requirement was modelled to check whether the *LSM* always performs a lane change to the same lane as requested by the *Planner*. This requirement was shown to be violated. Under certain circumstances the vehicle could indicate to go to the right (say), and check for traffic on the right side, but when it was clear to move into the right lane, the vehicle moved to the left. In our work, we further strengthen the property to express definite unsafe behaviours and the strengthened requirement is shown as *Req.1*.

***Req.1*:** If changing lane, the lane change shall always be to the same side as indicated.

In the following sections, we describe how formal verification is performed to show correctness of the *LSM* and to identify the violation of *Req.1* in the three different methods discussed in this paper. While there are several tools and tool based methods that support formal verification [11], [12], the choice of the tools discussed in this paper is primarily motivated by prior case studies with Supremica [17], [18], TLA$^+$ [19], [20], and SPARK [21], [22] on software systems similar in nature and scale to autonomous driving systems.

3 Supervisory Control Theory

The Supervisory Control Theory [23] (SCT) provides a framework for modelling, synthesis, and verification of reactive control functions for *discrete event systems* (DES), which are systems that occupy at each time instant a single *state* out of its many possible ones, and transits to another state on the occurrence of an *event*. Given a DES model of a system to control, the *plant*, and a *specification*[1] of the desired controlled behaviour, the SCT provides means to synthesize a *supervisor* that interacting with the plant in a *closed-loop* dynamically restricts the event generation of the plant such that the specification is satisfied.

Though the original SCT focused on synthesising supervisors that by construction fulfil the desired properties, a dual problem of interest here is to, given a model of a plant and specification, verify whether the specification is fulfilled or not. So, in this paper we use ideas from SCT to formally verify *LSM*, and do not focus on the synthesis of supervisors.

A DES modelling formalism appropriate in our context is finite-state machines extended with bounded discrete variables, with guards (logical expressions) over the variables and actions that assign values to the variables on the transitions [6].

Definition 1: *An Extended Finite State Machine (EFSM) is a tuple* $E = \langle \Sigma, V, L, \rightarrow, L^i, L^m \rangle$*, where* Σ *is a finite set of events,* V *is a finite set of bounded discrete variables,* L *is a finite set of locations,* $\rightarrow \subseteq L \times \Sigma \times G \times A \times L$ *is the conditional transition relation, where* G *and* A *are the respective sets of guards and actions,* $L^i \subseteq L$ *is the set of initial locations, and* $L^m \subseteq L$ *is the set of marked locations.*

The current state of such an *Extended Finite State-Machine* (EFSM) is given by its current location together with the current values of the variables. Thus, the state of an EFSM is not necessarily explicitly enumerated, but can be represented symbolically. This richer structure, though with equal expressive power, shows good modelling potential compared to ordinary finite state machines. The expression $l_0 \xrightarrow{\sigma:[g]a} l_1$ denotes a transition from location l_0 to l_1 labelled by event $\sigma \in \Sigma$, and with guard $g \in G$ and action $a \in A$. The transition is enabled when g evaluates to $\mathbf{T}$, and on its occurrence a updates

[1] In the SCT framework, the *specification* is the property of interest to verify with respect to the *plant*.

some of the values of the variables $v \in V$, thereby causing the EFSM to change location from l_0 to l_1.

EFSMs naturally interact through shared variables, but they can also interact through shared events, which is modelled by *synchronous composition*, where common events occur simultaneously in all interacting EFSMs, or not at all, while non-shared events occur independently. By this interaction mechanism a supervisor restricts the event generation of the plant; if the supervisor has a specific event in its alphabet but has no enabled transition labelled by that event from its current state, then the closed-loop system cannot execute that event in the current global state. We denote the synchronous composition of two EFSMs E_1 and E_2 by $E_1 \parallel E_2$ [6]. As defined by [6], transitions labelled by shared events but with mutually exclusive guards, or conflicting actions can never occur.

3.1 Nonblocking Verification

Given a set of EFSMs $\mathcal{E} = \{G_1, \ldots, G_n, K_1, \ldots, K_m\}$ where the components G_i $(i = 1, \ldots, n)$ represent the plant, and K_j $(j = 1, \ldots, m)$ represent the specification, we now want to determine whether the synchronous composition over all the components can from any reachable state always reach some marked state. The straightforward way to do this, called the *monolithic* approach, is intractable for all but the smallest systems, due to the combinatorial state-space explosion problem. Thus, more efficient approaches are needed.

One such approach that pushes the limit of what is tractable is the *abstraction-based compositional verification* [24], which has shown remarkable efficiency and manages to handle systems of industrially interesting sizes and complexity. It can be shown [24] that when $\mathcal{E}$ is blocking, this is due to some *conflict* between the components of $\mathcal{E}$. Thus, the approach of [24] employs *conflict-preserving abstractions* to iteratively remove redundancy and thus to keep the abstracted system size manageable. However, this approach eventually ends up converting the resulting abstracted EFSM system into ordinary finite-state machines, and then doing a monolithic verification of that. This then requires an efficient *explicit* verification algorithm, such as the one presented in [25].

3.2 Verification of *LSM* in Supremica

The software tool SUPREMICA [18] implements the nonblocking verification algorithms mentioned above (as well as various other algorithms, both for verification and synthesis). To verify whether *LSM* presented in Section 2 fulfils *Req.1* or not, we transform *Req.1* into an EFSM specification in such a way that with an EFSM model of the *LSM* code as the plant, the system will be nonblocking if and only if *LSM* fulfils *Req.1*.

The manual modelling of the *LSM* as an EFSM, similar to [17], is illustrated with a small excerpt from the actual MATLAB-code, shown in Listing 7.1 with some variable and state names anonymized. Listing 7.1 is a piece of the code that assigns variables and decides whether the system transits to a new state or not. The EFSM corresponding to the code is shown in Fig. 2. As described in Section 2, the *LSM* involves three execution stages during each call. The event *update* in the EFSM signifies the first stage: update on the inputs. The event *update* is followed by three transitions to model the possibility for the input variable `laneChangeRequest` to take one of the three values equally likely. Modelling the rest of the lines of code is straightforward. Note that the illustration provided is a minimal example to explain the modelling approach undertaken to manually model the *LSM* source code as an EFSM in SUPREMICA.

Listing 7.1: An illustrative excerpt from *LSM* code used for verification.

```
function duringStateA(var, laneChangeRequest)

    var.direction = laneChangeRequest;
    var.x = false;
    var.y = false;
    if laneChangeRequest != NoRequest
        var.state = StateB;
    end

end
```

Req.1 modelled as an EFSM is shown in Fig. 3. The event enterFinished denotes that the *LSM* has reached *State_Finished* completing the lane change process. The guard on the event checks for equality between two variables, `Output_Indication` and `Output_ChangeLane`. When these variables differ, the EFSM transits to a blocking state as shown in Fig. 3. `Output_Indication`

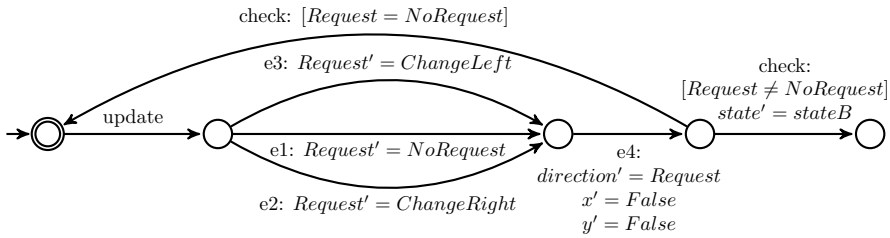


Figure 2: EFSM of Listing 7.1. Primed variables represent next-state values.

and `Output_ChangeLane` are modelled in a way such that they are set only during specific modes during the lane change process and are reset only when the *LSM* transits back to the initial state, when no lane change is requested. This makes it possible for their use in expressing *Req.1*. Modelling the *LSM* code in SUPREMICA resulted in an EFSM with 76 locations, 113 events, 144 transitions, and 20 variables. The synchronisation of the *LSM* with the EFSM in Fig. 3 resulted in a model with 1,522,117 reachable states, 113 events, and 2,164,607 transitions. The nonblocking verification of the synchronised model took less than a second and showed that a blocking state can indeed be reached. SUPREMICA also provides a 43 events long counter example that can be analysed in detail to understand the underlying cause.

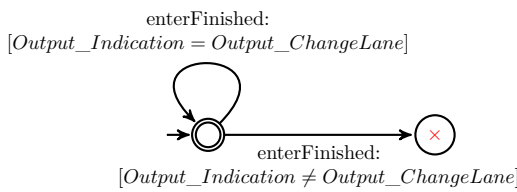


Figure 3: EFSM of the specification to model *Req.1*. The blocking state is represented with a cross inside.

4 Model Checking

Model checking [26], [27] is a framework for verification of finite transition systems using temporal logic [28] as a specification formalism. Several for-

malisms and powerful model checking tools have emerged over the years [29], [30].

Definition 2: *A finite transition system is a tuple* $\mathcal{T} = \langle S, Act, \rightarrow, I, AP, L \rangle$ *where* S *is a finite set of states,* Act *is a finite set of actions,* $\rightarrow \subseteq S \times Act \times S$ *is a transition relation,* AP *is a finite set of atomic propositions, and* $L : S \rightarrow 2^{AP}$ *is a labelling function.*

Given a transition system $\mathcal{T}$, and a temporal logic formula f, the model checking problem is a decision procedure for $\mathcal{T} \vDash f$. If $\mathcal{T} \nvDash f$, then the model checking algorithm provides a counter example as an evidence for the violation, which can then be used to analyse the issue and the ways to resolve it.

4.1 Temporal Logic of Actions

The Temporal Logic of Actions (TLA) is a logical formalism for specifying and reasoning about concurrent systems [31]. TLA is a variant of temporal logic [28] and uses the notion of states and actions to model behavioural properties of systems. TLA, as a logical formalism provides the expressive power to reason about programs using assertions on states and pairs of states (actions). Actions are predicates that relate two consecutive states and are used to capture how the system is allowed to evolve. This section only presents a brief overview of TLA and the associated formalism for specifying and model checking systems. A more detailed description of the language and other advanced advanced topics is available in [7], [19], [31].

The reasoning system in TLA is built around TLA formulas. A TLA *formula* is true or false on a behaviour. A *behaviour* in TLA is an infinite sequence of states. A *state* in TLA is an assignment of values to variables and a *step* is a pair of states. Steps of a behaviour denote successive pairs of states. Given a system S, with the executions of the system represented as behaviours, and a formula f, we can decide whether S satisfies f iff the formula f is true for every behaviour of S.

The elementary building blocks of a TLA formula include state predicates, actions, logical operators (such as $\wedge$,$\neg$, etc.), the temporal operator $\Box$ (always) and the existential quantifier $\exists$. A *state predicate* is a boolean valued expression (predicate) on states. An action, $\mathcal{A}$, is a boolean valued expression (predicate) on steps. Actions are formed from unprimed variables and primed variables to represent the relation between old states and new states. The

unprimed variables refer to the values of the variables in old states, the first state of the step, whereas the primed variables refer to the variable values in new states, the second state of the step. State predicates have no primed variables. A step is an $\mathcal{A}$-step if it satisfies $\mathcal{A}$. An action is valid, $\vDash \mathcal{A}$, iff every step is an $\mathcal{A}$-step. In TLA, atomic operations of programs are represented by actions.

TLA$^+$ is a formal specification language based on formal set theory, first order logic and TLA. A TLA$^+$ specification, typically denoted *Spec*, is a temporal formula predicate on *behaviours*. All the behaviours satisfying *Spec* constitute the correct behaviours of the system. TLA$^+$ describes a system as a set of behaviours with an initial condition and a next state relation. The initial condition specifies the possible initial states and the next state relation specifies the possible steps. A TLA$^+$ specification is a temporal formula of the form

$$Spec \triangleq Init \wedge \Box[Next]_{\langle vars \rangle} \wedge Temporal \tag{A.1}$$

where *Init* is a state predicate corresponding to the initial condition, *Next* is an action corresponding to the next state relation, *vars* is a tuple of all variables in the specification, and *Temporal* is a temporal formula usually specifying liveness conditions. Formula *Spec* can be seen as a predicate on behaviours. *Spec* is true for a behaviour σ, iff *Init* is true in the first state of σ and every step in σ is either a step that satisfies *Next* or is a *stuttering* step. A *stuttering* step is one in which none of the variables are changed.

The specification (A.1) can be model checked using the TLC model checker. TLC takes a TLA$^+$ specification and checks whether the specification satisfies the desired properties by evaluating all possible behaviours of the specification. The TLA$^+$ specification language accompanied by an IDE consisting of TLC and other useful tools can be downloaded from [19].

4.2 Verification of *LSM* in TLA$^+$

The approach we use to formally verify the *LSM* in TLA$^+$ is similar to the approach of Supremica. The *LSM* code is manually translated in TLA$^+$ using the constructs available in the specification language. Listing 7.2 shows the TLA$^+$ translation of the MATLAB-code in Listing 7.1 as a TLA$^+$ formula that relates unprimed variables and primed variables using arithmetic and logical operators. The formula describes the allowed behaviour of the function in

Listing 7.1. A call to the function `duringStateA` is translated to a behaviour where the formula `During_StateA` is valid.

Listing 7.2: TLA$^+$ translation of the code in Listing 7.1.

```
1  During_StateA ==
2             /\ Lane_Change_Request' \in ...
                   {"NoRequest","ChangeLeft","ChangeRight"}
3             /\ var_state = "StateA"
4             /\ var_direction' = Lane_Change_Request
5             /\ var_x' = FALSE
6             /\ var_y' = FALSE
7             /\ IF Lane_Change_Request # "NoRequest" THEN
8                   var_state' = "StateB"
9                ELSE UNCHANGED var_state
```

The TLA$^+$ translation of the entire *LSM* code consists of an initial state predicate, *Init* and *Next*. *Next* is composed of smaller sub-formulae, each corresponding to different functions in the original code, of which one formula is shown in Listing 7.2. With the complete TLA$^+$ translation of the *LSM*, TLC can model check for desired properties, which are described using predefined statements and constructs available. More details on the statements and the restrictions on TLC is available in [7]. In order to verify *Req.1* of Section 2, we make use of invariant checking in TLC.

An *invariant*, typically denoted as *Inv*, of a *Spec* is a state predicate that should be valid in all reachable states. Invariants can be defined for specifications as well as next-state actions. An invariant of a specification that is also an invariant of a next-state action is sometimes called an inductive invariant of *Spec*. In model checking mode for invariance checking, TLC explores all reachable states and looks for states in which the invariant is not satisfied.

Req.1 is translated to a TLA formula as

$$\text{InvProp} \triangleq \neg(\text{var\_state} = \text{"State\_Finished"} \\ \wedge \text{Output\_Indication} \neq \text{Output\_Change\_Lane}). \quad \text{(A.2)}$$

Reaching a state where InvProp is violated means that the state predicate evaluate to false, i.e. a behaviour where the lane change is finished and the outputs for showing indication and changing lane differ, is allowed in our specification, thereby showing the presence of an error in our code. The

complete TLA$^+$ translation was 250 lines with 20 variables. In model checking mode using breadth-first search, TLC shows the violation of InvProp with a 5 step long error trace for analysis.

5 Deductive Verification

Model checking is well suited to establish (temporal) properties of state traces, but mostly requires *abstractions* over the real source code. In contrast to that, deductive verification [32] techniques are well suited for fully precise reasoning about the computation on the *source code level*. Often, first order-logic is used to characterise conditions on the data in specific states, in pre and post-conditions of procedures, or invariants. Deductive verification typically uses a compositional methodology, specifying and verifying one procedure at a time. Verification tools exist for common programming languages such as C [33], Java [34], or Ada [22].

5.1 SPARK

Ada [35] is a high level imperative programming language targeting the development of large scale safety critical software. Ada is suited to meet the high integrity software requirements and has been used in several industrial embedded software development projects [21]. SPARK is a subset of Ada with additional features to support formal verification [22]. SPARK uses property specifications in the form of program annotations described inline with the source code to perform static program analysis and build automated proofs to show the correctness of the software. In that sense, SPARK uses the correct by construction philosophy through contract based programming to develop software.

A SPARK program is made up of one or more program units. Subprograms and packages are two examples of SPARK program units. A subprogram execution is invoked by a call and subprograms express a sequence of actions. Procedures and functions are the two types of subprograms in SPARK. Procedure calls are standalone statements, whereas function calls occur in an expression and return a value. Packages group together entities like data types, subprograms, etc., and can be considered to be the equivalent of header files in an object oriented programming language like C++. A program unit consists

of two structures, a specification and a body. The specification contains the variables, types and the subprogram declarations with their annotations. The body of a program unit contains the details of the implementation.

Properties are in SPARK specified using subprogram contracts (pre and post-conditions), loop invariants, and data dependencies. The formal verification toolset in SPARK can perform program analysis on the source code at various levels. Flow analysis capabilities ensure the program correctness with respect to data flow and information flow. Errors arising due to uninitialized variables, data dependencies between inputs and outputs of subprograms, well-formedness of programs, etc., are checked by this level of analysis. A higher level of analysis is to perform automated proofs to check for run time errors and conformance of the program with the specifications. The program annotations specified are used to generate *verification conditions*, which can then be discharged using the proof tools to show program correctness.

5.2 Verification of *LSM* in SPARK

SPARK 2014 [21] and its associated tools are used to formally verify the *LSM*. With the use of packages and subprograms in SPARK, the code structure of the original implementation of *LSM* using classes and methods in MATLAB-code is preserved. Listing 7.3 shows how the code in Listing 7.1 is built in SPARK. The implementation is done as a procedure (subprogram). Lines 1-6 represent the specification part of the subprogram and lines 8-19 represent the body. The specification consists of the subprogram declaration and its contract in the form of pre and post-conditions. The parameter mode `in ... out` permits both read and write operations on the values of the associated parameter.

SPARK has a set of core annotations as predefined rules that can be checked without user defined contracts. However, here we are interested in verifying functional properties like *Req.1* and therefore SPARK needs stronger annotations to perform formal analysis. The contract specified in Listing 7.3 is an illustrative example of type of contracts used to show correctness of *LSM* with respect to *Req.1*. The preconditions, denoted `Pre`, are assertions that are satisfied when the procedure is called and the postconditions, denoted `Post`, are the conditions that should be satisfied as a result of the procedure call. These contracts are used by the analysis tools to generate verification conditions, which are mathematical expressions relating a number of hypotheses

Listing 7.3: SPARK implementation of the code in Listing 7.1.

```
1  procedure During_StateA
2      (Var                  : in out Var_Type;
3       Lane_Change_Request : in Lane_Change_Direction_Type)
4  with Pre  => Var.State = StateA,
5       Post => ((Var.Direction = Lane_Change_Request) and
6                (Var.State in StateA | StateB));
7  ------------------------------------------------
8  procedure During_StateA
9      (Var                  : in out Var_Type;
10      Lane_Change_Request : in Lane_Change_Direction_Type)
11 is
12 begin
13      Var.Direction := Lane_Change_Request;
14      Var.X := False;
15      Var.Y := False;
16      if Lane_Change_Request /= NoRequest then
17         Var.State := StateB;
18      end if;
19 end During_StateA;
```

(obtained from preconditons) and conclusions (from postconditions). Providing a correctness proof of the program then boils down to showing that the conclusions always follow from the hypotheses. Detailed information on the the analysis tools is available in [22], [36].

With this general idea, the initial approach to prove correctness of the *LSM* was to specify one global contract to capture *Req.1*. This global contract was specified on the complete *LSM* code implemented as a package in SPARK. However, results from the analysis showed that one global contract was insufficient to show correctness of *Req.1*. Subsequent annotations were added to the different subprograms. *Req.1* was specified as a postcondition (A.3) of a subprogram responsible for execution on the completion of a lane change.

```
Post ⇒ (Var.State = Finished) and
                    (Output_Indication = Output_ChangeLane)   (A.3)
```

Although the proof checks for most of the subprogram contracts were automatically proved by SPARK analysis tools, error messages from proof checks reported that a few postconditions including (A.3) might fail. The unproved checks could possibly indicate incorrectness of the code (implementation and specification) or the need for stronger annotations for the tools in the form of

intermediate assertions and better code organisation. In order to conclusively decide the cause for the failed proof checks, more manual reviews, analysis of the execution paths corresponding to the failed checks and possibly stronger contracts were needed. However, the undertaken approach of implementing the code first and then incrementally annotating the subprograms in order to satisfy the property turned out to be inefficient. A better work flow in our case would be the reverse approach, where the property is formally broken down into suitable subprogram contracts followed by the implementation to show correctness.

6 Insights and Discussion

This section provides a discussion and the insights gained from this case study. The discussion is focused on how the verification methods aid in addressing the challenges mentioned in Section 1, and does not aim to compare the performances or the algorithms of the tools.

Describing the system. Autonomous driving systems are often categorised as Cyber-Physical Systems (CPS) or reactive systems in literature, depending on the focus of research. Irrespective of the classification, modelling and observing the system and its *environment* is a known challenge. The expressive power is limited to the choice of formalism. In our case, describing *LSM* as extended finite state machines and transition systems (although not too different) was sufficient to capture—and reason about—correctness due to its discrete nature. However, correctness of *Path Planner*, *Controller*, *Sense* in Fig. 1 is just as crucial as *LSM* and the formalism discussed in this paper might not be sufficient as they have continuous dynamics and probabilistic behaviour. Choosing task specific formalisms and tools for different software development teams complicates the industrial adoption of such techniques. In this respect, having subtle and necessary extensions to the existing formalisms so as to capture a wider spectrum of abstractions, while still being decidable, can be invaluable.

Modelling the observable behaviour of the environment faces the risk of state-space explosion. Defining the operating boundaries of the environment with respect to the system is very crucial in successfully addressing the challenge. For example, in our case of the lane change software module, the traffic

state (position, behaviour of other vehicles,...) could serve as a definition of the environment for the decision making component in Fig. 1. However, using the same definition for environment to model and reason about *LSM* or *Path Planner*, would neither help tackle the challenge nor be an efficient use of any of the formal technique discussed in this paper. The use of deductive verification in SPARK decouples from such problems by applying verification techniques on the source code. Nevertheless, the challenge then manifests in the need to write complex functional specifications to have the formal analysis done, as it turned out in our case.

From our experience, the key to address these challenges is to use formal approaches with different levels of abstractions to *divide and conquer* in a modular way, similar to classical large scale software development. Higher level abstractions could be used to define logical boundaries between the systems and their environments and lower level abstractions to reason about the systems within their boundaries. Compositional verification can then be used to reason about systems in a modular way. SUPREMICA, TLA$^+$ and SPARK have features to support such compositional verification of systems. This work flow could also be used to formally obtain subprogram annotations in the deductive verification framework to show correctness of source code.

Requirements and Properties. In this paper, the focus is to verify one requirement that affects the safety of the system. In the SCT framework, EFSM is used as the specification language. A violation of the requirement is modelled as an event leading to a blocking state and nonblocking verification is performed to check for errors. This is similar to checking whether in all computations, we eventually reach a state from where a marked state can be reached. While nonblocking cannot be directly translated in linear-time temporal logic, the use of invariants is exploited in TLA$^+$ to check for the desired property. In SPARK, the use of pre/post conditions to look for the particular unsafe behaviour did not prove to be an efficient work method. While TLA$^+$ and SUPREMICA provided counter examples that could help in the analysis of the bug, the counter example generation in SPARK was not sufficient to draw concrete conclusions in our particular case. This could be attributed to the fact that for efficient use of automated reasoning in contract based programming, operational completeness, meaning contracts for normal, error and exceptional behaviour should be included in the specification. The

reverse approach of implementing the source code first and then annotating with contracts to check for a particular unsafe behaviour proved very inefficient. However, a program crashing is just as unsafe as compared to the behavioural safety property discussed in this paper. For such software program malfunction due to run time errors (such as division by zero, overflow, etc.), modelling and specifying in SUPREMICA and TLA$^+$ is complicated and will greatly increase the complexity. SPARK is efficient in this regard.

Type of analysis and the scope of correctness. Formal methods can be applied to all levels of the software development process. While acknowledging the individual strengths of each of the methods discussed in this paper, no method on its own is sufficient to prove correctness for the *LSM*. Supervisory control and TLA$^+$ are abstract methods that are best suited for verification at the system level, software architectural level and software design level of the ISO 26262 standard. Deductive verification methods give the most benefit at the software unit (program) verification, the lowest level (source code) of the V-model. SPARK is developed to suit the needs of high integrity safety critical applications and therefore provides better evidence for compliance to several clauses of the standard at the software unit verification level. The abstraction based approaches discussed in this paper involves manual modelling of the system and therefore requires additional effort to ensure that the right detail is captured in the modelling as well as in specifying the properties. The occurrence of false alarms in such methods is of course an implicit trade-off.

Leveraging formal methods in an industrial setting. The verification approaches discussed in this paper are all performed after the software was implemented. A software to solve an intended function was written in a programming language and then verified for correctness. Although, better use of the methods described in this paper could be made in the earlier stages of the development process (correct by construction approach), the situation where software is verified for correctness in the later stages seems more common in the industrial setting. In our experience, the challenging task encountered while working with the abstract methods is the lack of interoperability with the other tools used in the development. SUPREMICA and TLA$^+$ are stand alone methods and currently, the only way to use them is for engineers to have parallel activities, one with the formal tools and the other with the con-

ventional development tools. While this might be justified for high integrity applications, the need for manual effort to synchronise the parallel activities to obtain a concrete impact is often a drawback. Work on suitable intermediary plug-ins to have traceability between the informal requirements management activity and the formal specification methods would definitely work in favour of increased adoption in the software specification stages. Counter-example generation in the abstract methods discussed in this paper is easily the highest return on investment in an industrial setting. This could further be enhanced by work on using counter-examples to generate test scenarios in the preferred testing framework in the development routine. This will also suit well within the continuous development and continuous integration principles of agile development. In this regard, SPARK is well suited for easier integration. However, the use of SPARK as an after development verification tool without formal specification in the earlier stages, is still inefficient.

7 Conclusion

In this paper, we have applied formal verification based on Supervisory Control Theory, Model Checking and Deductive Verification to verify correctness of a decision making software in an autonomous vehicle. Discussion on how the verification scenario differs in each of the methods is presented. We also provide insights on how the different approaches can address the challenges in industrial development of safe autonomous driving software. The difficulty in working with all these tools is not in learning them but in capturing the right level of abstraction for the verification objectives and stating the formal properties. Although this paper deals with the verification of one safety requirement of a decision making software module, the insights gained are valuable to address the challenges. Future work includes the investigation of integrating multiple formal approaches to tackle the challenges mentioned in this paper also to scale the approaches to different types of systems in an autonomous vehicle for larger classes of properties with more software requirements.

References

[1] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey", *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, 2017.

[2] M. Luckcuck, M. Farrell, L. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey", *arXiv preprint arXiv:1807.00048*, 2018.

[3] ISO, "Road vehicles – Functional safety", Tech. Rep. ISO 26262, 2011.

[4] K. Forsberg and H. Mooz, "The relationship of system engineering to the project cycle", in *INCOSE International Symposium*, Wiley Online Library, vol. 1, 1991.

[5] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes", *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[6] M. Skoldstam, K. Akesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables", in *2007 46th IEEE Conference on Decision and Control*, IEEE, 2007, pp. 3387–3392.

[7] L. Lamport, *Specifying systems: the TLA$^+$ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[8] K. R. Apt, F. S. de Boer, and E. Olderog, *Verification of Sequential and Concurrent Programs*, ser. Texts in Computer Science. Springer, 2009.

[9] S. A. Seshia, D. Sadigh, and S. S. Sastry, "Formal methods for semi-autonomous driving", in *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, IEEE, 2015.

[10] M. Fisher, L. A. Dennis, and M. P. Webster, "Verifying autonomous systems.", *Commun. ACM*, vol. 56, no. 9, pp. 84–93, 2013.

[11] R. C. Armstrong, R. J. Punnoose, M. H. Wong, and J. R. Mayo, "Survey of existing tools for formal verification", *SANDIA REPORT SAND2014-20533*, 2014.

[12] B. Beckert and R. Hähnle, "Reasoning and verification: State of the art and current trends", *IEEE Intelligent Systems*, vol. 29, no. 1, pp. 20–29, 2014.

[13] R. Kasauli, E. Knauss, B. Kanagwa, A. Nilsson, and G. Calikli, "Safety-critical systems and agile development: A mapping study", in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, IEEE, 2018.

[14] R. A. Kemmerer, "Integrating formal methods into the development process", *IEEE software*, vol. 7, no. 5, pp. 37–50, 1990.

[15] H. Saiedian and M. G. Hinchey, "Challenges in the successful transfer of formal methods technology into industrial applications", *Information and Software Technology*, vol. 38, no. 5, pp. 313–322, 1996.

[16] S. Wolff, "Scrum goes formal: Agile methods for safety-critical systems", in *Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, IEEE Press, 2012, pp. 23–29.

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

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

[19] L. Lamport, *The TLA$^+$ home page*, `https://lamport.azurewebsites.net/tla/tla.html`, Accessed: 2019-04-22.

[20] C. Newcombe, "Why amazon chose TLA$^+$", in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Springer, 2014, pp. 25–39.

[21] *Adacore - homepage*, `https://www.adacore.com/`, Accessed: 2019-04-26.

[22] J. Barnes, *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, 2012.

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

[24] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional nonblocking verification of extended finite-state machines", *Discrete Event Dynamic Systems*, vol. 26, no. 1, pp. 33–84, 2016.

[25] R. Malik, "Programming a fast explicit conflict checker", in *2016 13th International Workshop on Discrete Event Systems (WODES)*, IEEE, 2016, pp. 438–443.

[26] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic", in *Workshop on Logic of Programs*, Springer, 1981, pp. 52–71.

[27] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR", in *International Symposium on programming*, Springer, 1982, pp. 337–351.

[28] A. Pnueli, "The temporal logic of programs", in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, IEEE, 1977, pp. 46–57.

[29] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[30] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018.

[31] L. Lamport, "The temporal logic of actions", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.

[32] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 583, Oct. 1969.

[33] N. Kosmatov, V. Prevosto, and J. Signoles, "A lesson on proof of programs with Frama-C. Invited tutorial paper", in *Tests and Proofs*, M. Veanes and L. Viganò, Eds., Springer, 2013, ISBN: 978-3-642-38916-0.

[34] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification—The KeY Book*, ser. LNCS. Springer, 2016, vol. 10001.

[35] Barnes, John, *Programming in Ada 2012*. Cambridge University Press, 2014.

[36] *Spark 2014 reference manual*, `https://docs.adacore.com/spark2014-docs/html/lrm/index.html`, Accessed: 2019-04-26.

PAPER B

Supervisory Control Theory in System Safety Analysis

Yuvaraj Selvaraj, Zhennan Fei, Martin Fabian

Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops,
Lecture Notes in Computer Science vol. 12235, pp. 9–22, 2020.
 DOI: 10.1007/978-3-030-55583-2_1

The layout has been revised.

Abstract

Development of safety critical systems requires a risk management strategy to identify and analyse hazards, and apply necessary actions to eliminate or control them as malfunctions could be catastrophic. Fault Tree Analysis (FTA) is one of the most widely used methods for safety analysis in industrial use. However, the standard FTA is manual, informal, and limited to static analysis of systems. In this paper, we present preliminary results from a model-based approach to address these limitations using Supervisory Control Theory. Taking an example from the Fault Tree Handbook, we present a systematic approach to incrementally obtain formal models from a fault tree and verify them in the tool Supremica. We present a method to calculate minimal cut sets using our approach. These compositional techniques could potentially be very beneficial in the safety analysis of highly complex safety critical systems, where several components interact to solve different tasks.

1 Introduction

Software development in safety critical systems necessitates a risk management strategy to identify and analyse risks, and to apply the necessary actions to eliminate or control them. The objective of safety analyses, performed during various development phases, is to ensure that the risk of safety violations due to the occurrence of different faults is sufficiently low.

Fault Tree Analysis, FTA [1], is one of the most common methods for safety analysis in various industries. While standard fault trees are simple and informative, they are not free from limitations [2]. Standard FTA is primarily a manual process based on an informal model, i.e., the process relies on the system analysts and domain experts to systematically think about all risks and their possible causes. The lack of formal semantics makes it difficult to verify the correctness of the safety analysis, especially for rapidly evolving industries like the autonomous driving industry where new edge cases are continuously identified. In complex industrial software controlled systems, safety models

must capture many possible interactions between system components, where different interleavings of failure events can either result in a failure or operational state. Standard fault trees are not suitable for modelling temporal, sequential and state dependencies of events. Another notable shortcoming with standard FTA for large and complex systems is the need for safety analyses to be intuitive and compositional. This is crucial in projects where the system of interest comprises interacting sub-systems, possibly delivered by different teams or suppliers.

Though several limitations exist, FTA is one of the widely used safety analysis methods. Different extensions to standard fault trees [3] have been proposed to address some of the limitations. Research on using formal logic in FTA [4]–[6] address the limitation of informal and manual FTA process. Extensions like dynamic fault trees [7], state-event fault trees [8], and temporal fault trees [9] address inability of standard fault trees to model dynamic behaviour. The most widely used extension to include temporal sequence information is dynamic fault trees [3], [7]. Over the years, research on the development of model-based dependability analysis (MBDA) [10] techniques have enabled automated dependability analysis. In [10], such emerging MBDA techniques are classified into two paradigms. The first paradigm, termed failure logic synthesis and analysis focuses on automatic construction of failure analyses and the second paradigm, termed behavioural fault simulation focuses on formal verification based techniques. Despite this research, challenges remain in addressing the limitations with standard fault trees and safety analysis [3], [10]. Thus any progress in addressing these limitations is helpful. The preliminary results presented in this paper is part of an ongoing endeavour to address the aforementioned limitations by a model-based approach based on Supervisory Control Theory (SCT) [11].

The formal models used in the SCT framework can describe dynamic behaviour, which is often needed to analyse modern and complex safety critical systems. The compositional abstraction based algorithms used in SCT allow automated synthesis and verification of safety models for large and complex systems. These features of the SCT framework makes it possible to define a complete model-based safety analysis approach with automated analysis. To ensure sufficient detail of explanation and some degree of familiarity, we do not present a complex example in this paper; instead we describe our approach using a rather simple example from the *Fault Tree Handbook* [1].

We make three main contributions in this paper. First, we address the issue of informal description of standard fault tree analysis by presenting a systematic approach to incrementally obtain formal models from a fault tree. Second, we present a method to analyse the fault trees using the SCT tool Supremica [12]. Finally, we present a method to calculate minimal cut sets using our approach. An advantage of our work is the compositional approach to modelling and verification that is beneficial in reasoning about large fault trees for highly complex systems. To the best of our knowledge, SCT has not previously been used in the context of fault tree analysis.

The paper begins with a brief introduction to FTA and SCT in Section 2 and Section 3, respectively. Section 4 discusses modelling and analysis in Supremica with an example from the *Fault Tree Handbook* [1]. The paper is concluded with a brief discussion on future extensions in Section 5. Our work is successfully integrated with a model-based systems engineering tool [13], that is widely used in the automotive industry.

2 Fault Tree Analysis

Fault Tree Analysis (FTA) [1] is a top-down deductive safety analysis technique, where an undesired safety-critical failure of a system is specified, and then analysed in the context of its operational environment to find all possible ways in which the specified failure can occur.

A fault tree is a graphical model of various combinations of faults that cause the safety critical failure, represented as a top level failure event at the root of the fault tree. From this root event, the fault tree is constructed from a predefined set of symbols [1], which results in a set of combinations of component failures that can cause the top level failure. Note that the fault tree is not a model of all possible causes for system failure, but given a particular failure it depicts the possible combinations of basic component failures that lead to this failure. Since FTA is primarily a manual process, the exhaustiveness of the analysis is left to the assessment of the analyst.

Although several extensions of fault trees have been proposed [3], in this paper we limit ourselves to the symbols described in the *Fault Tree Handbook* [1]. Broadly, the nodes in the fault tree can be classified into three types: *events*, *gates*, and *transfer symbols* [1].

2.1 Pressure Tank System

The pressure tank system [1] in Fig. 1 describes a control system to regulate a pump-motor that pumps fluid into the tank. Initially the system is considered to be dormant and de-energized: switch S1 open, relays K1 and K2 open, and the timer relay closed. The tank is assumed to be empty in this state and therefore the pressure switch S is closed. It is also assumed that it takes 60 seconds to pressurize the tank, and an outlet valve, which is not a pressure relief valve, is used to drain the tank.

System operation is started by pressing switch S1. This closes and latches relay K1, and subsequently relay K2 to start the pump. When threshold pressure is reached, the pressure switch opens, causing K2 to open, and consequently the pump motor to cease operation. The timer allows emergency shut-down in case the pressure switch fails. Initially, the timer relay is closed and power is applied to the timer as soon as K1 closes. If the clock in the timer registers 60 seconds of continuous power, the timer relay opens and latches, thereby causing a system shut-down. In normal operation, when pressure switch S opens, the timer resets to 0 seconds. When the tank is empty, the pressure switch closes, and the cycle can be repeated.

Fig. 2 shows the basic fault tree from [1] (page VIII-13) for the pressure tank system. Here, the hazard 'rupture of pressure tank after start of pumping' is analysed and is represented by the top level failure event, E1. The basic events denoted by circles represent the respective component failures and form the leaves of the tree. The intermediate events, which are fault events that occur due to one or more antecedent causes are denoted by rectangles. The process of obtaining the fault tree following a top down analysis is out of scope of this paper; we assume a FT is given. A complete description of the example and the fault tree can be found in [1].

3 Supervisory Control Theory

The Supervisory Control Theory (SCT) [11] provides a framework to model, synthesize and verify control functions for *discrete event systems* (DES), which are dynamic systems characterised by the evolution of events causing the system to transit from one discrete state to another. Given a model of the system to control, a *plant*, and a *specification* describing the desired controlled behaviour, the SCT provides methods to synthesise a *supervisor* that

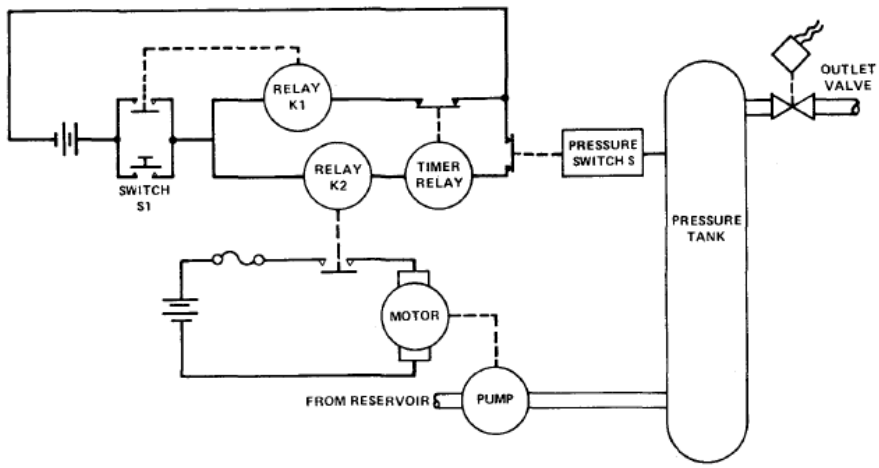


Figure 1: Pressure Tank System from [1], page VIII-1

dynamically interacts with the plant in a closed-loop, and restricts the event generation of the plant such that the specification is satisfied. The supervisor thus ensures a safe control of the plant by restricting the execution of certain events. However, only events that are *controllable* can be restricted by the supervisor, while events that are *uncontrollable* cannot be restricted. A dual problem that is of interest here, is to given a model of a (controlled) plant and a specification, *verify* whether the specification is fulfilled or not. So, in this paper we use ideas from SCT to formally verify properties of the plant model, and do not focus on the synthesis of supervisors.

To model a fault tree as a DES, we use Extended Finite State Machines (EFSM) [14], which are finite state machines extended with bounded discrete *variables*, *guards* that are logical expressions over variables, and *actions* that assign values to variables on transitions.

Definition 1: *An Extended Finite State Machine (EFSM) is a tuple* $E = \langle \Sigma, V, L, \rightarrow, l^i, L^m \rangle$*, where* Σ *is a finite set of events,* V *is a finite set of bounded discrete variables,* L *is a finite set of locations,* $\rightarrow \subseteq L \times \Sigma \times G \times A \times L$ *is*

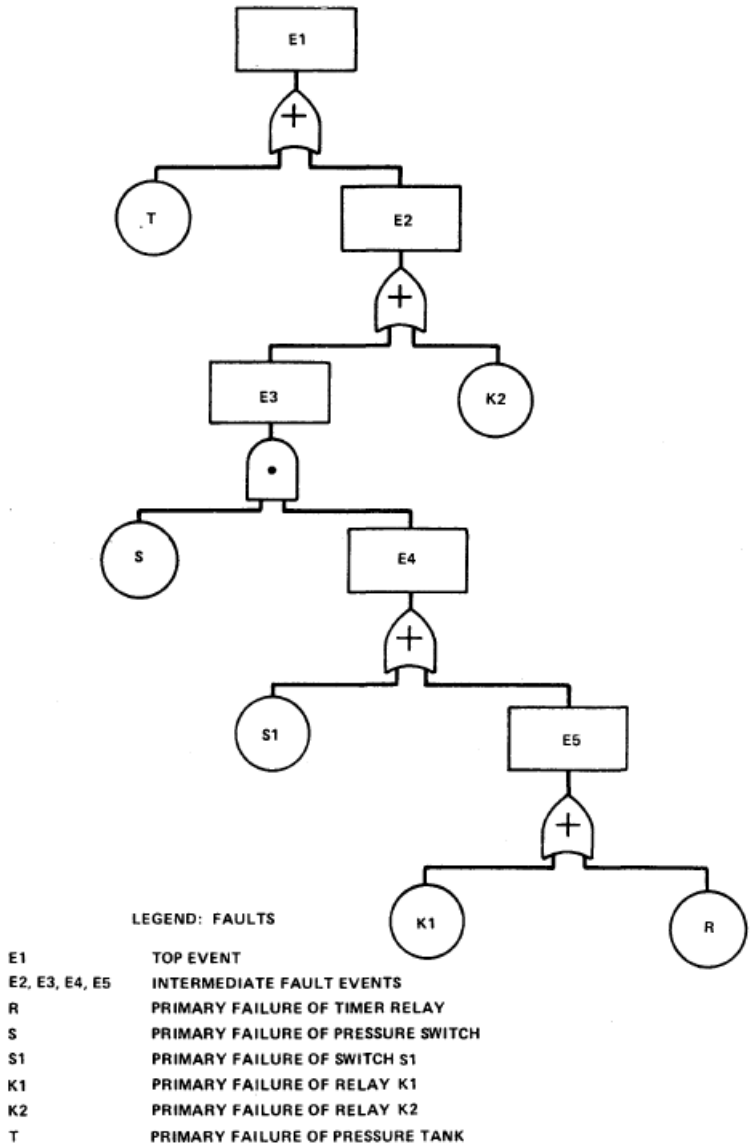


Figure 2: Fault Tree for Pressure Tank System in Fig. 1 from [1], page VIII-13

the conditional transition relation, where G and A are the respective sets of guards and actions, $l^i \in L$ is the initial location, and $L^m \subseteq L$ is the set of marked locations.

A state in an EFSM is given by its current location together with the current values of the variables. The expression $l_0 \xrightarrow{\sigma:[g]a} l_1$ denotes a transition from location l_0 to l_1 labelled by event $\sigma \in \Sigma$, with guard $g \in G$, and action $a \in A$. The transition is enabled when g evaluates to *true*, and on its occurrence, the current location of the EFSM changes from l_0 to l_1, while a updates some of the values of the variables $v \in V$. EFSMs interact through shared events by *synchronous composition*, denoted $\mathcal{A}_1 \parallel \mathcal{A}_2$ for two interacting EFSM models, $\mathcal{A}_1$ and $\mathcal{A}_2$. In synchronous composition, shared events occur simultaneously in all interacting EFSMs, or not at all, while non-shared events occur independently. Transitions on shared events with mutually exclusive guards, or conflicting actions will never occur [14]. In an EFSM, *active events* are the events that label some transition, while *blocked events* do not label any transition. In the synchronous composition of two EFSMs, the blocked events of the synchronised EFSM, is the union of the blocked events of the synchronised EFSMs. That is, transitions in one EFSM labelled by events blocked by the other EFSM, will be removed.

3.1 Nonblocking verification

Given a set of EFSMs $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$, the *nonblocking* property guarantees that some marked state can always be reached from any reachable state in the synchronous composition over all the components $\mathcal{A}_i$. While the monolithic approach to nonblocking verification is explicit, it is limited by the combinatorial state-space explosion. The *abstraction-based compositional verification* [15] has shown remarkable efficiency to handle systems of industrial complexity. This approach employs *conflict-preserving abstractions* to iteratively remove redundancy and keeps the abstracted system size manageable. SUPREMICA [16], a tool for modelling and analysis of DES models, implements the abstraction-based compositional algorithms (and others) for verification of EFSMs.

4 FTA in Supremica

In this section, we describe how the fault tree in Fig. 2 is modelled into a number of *plant* EFSMs. We demonstrate how the model can be validated by verifying typical *specifications* in SUPREMICA. This section also includes a brief discussion about computing minimal cut sets using our approach. Both SUPREMICA and the models of this section are available online[1].

4.1 Modelling

To make the best use of compositionality, we incrementally model different failure events in a modular way. Given a fault tree, we first model the lowest level and gradually proceed towards the top level event. For the higher levels, we only consider the intermediate fault events from the lower levels and hide all other inner details.

Consider the lowest level of the fault tree in Fig. 2. It consists of two basic events as inputs to the lowest OR gate leading to the intermediate fault event, E5. This forms the first level in our modelling hierarchy. Fault event E5 can occur either due to a primary failure of K1 or a primary failure of R. This behaviour is modelled in the EFSM as shown in Fig. 3a[2]. The two events $K1$ and R denote the corresponding primary failures and when either occurs, the EFSM transits from its initial location, A_0^i to location E_5.

With E5 modelled, we proceed to the next level, the intermediate fault event E4. From Fig. 2, we see that this can occur either due to a primary failure of switch S1 or due to the occurrence of E5. This gives us a total of 7 possible combinations that lead to E4. However, since we have modelled the analysis for E5 as an EFSM on the previous level, we can use guards to capture this, and model E4 with just 2 events as shown in Fig. 3b. The guard condition on the event $E5$ ensures that the event is enabled only in a situation where the EFSM in Fig. 3a is in location E_5. Here, the guard $[A_0 == E_5]$ represents that the current location of the EFSM A_0 in Fig. 3a, is E_5.

The next level in our modular hierarchy is the output event of the only AND gate in the fault tree, E3. The two inputs to the AND gate correspond to the primary failure of the pressure switch S and the analysis resulting from the intermediate fault E4. Fig. 4 shows the model for this fault event E3. Since

[1] https://supremica.org
https://github.com/yuvrajselvam/FTA_SCT

(a) EFSM modelling E5 **(b)** EFSM modelling E4

Figure 3: EFSMs for intermediate failure events, E4 and E5 of the fault tree

the order of events do not matter in an AND gate, there are two possible ways to reach the failure state as shown in Fig. 4.

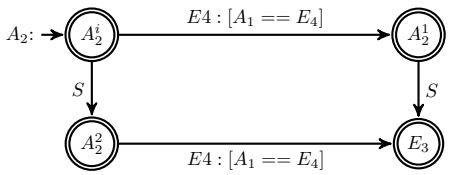


Figure 4: EFSM, A_2 for the intermediate failure event E3

The final two levels of the fault tree corresponding to fault events E2 and E1 consist of OR gates and are modelled as already shown, see Fig. 5. Note that in the plant models, the only unmarked location is the initial location in Fig. 5b, and therefore in the synchronised plant model, which gives the complete fault tree, the marked locations correspond to the top level failure event E1.

(a) EFSM modelling E2 **(b)** EFSM modelling E1

Figure 5: EFSM for intermediate failure events, E2 and E1 of the fault tree

For special cases of AND gates, like INHIBIT and PRIORITY-AND, the

[2]In this paper, for a fault Ex in the FT, Ex denotes the corresponding event in the EFSM and E_x denotes the location reached due to the occurrence of the fault.

models look slightly different. For an INHIBIT gate, where the output is determined by a single input together with some qualifying condition, we can use a single event label together with the qualifying condition as a guard to model the transition to the failure state. For a PRIORITY-AND gate, where the output occurs only if all inputs occur in a specified ordered sequence, we can model the specified sequence as a path from the initial state to the failure state. For example if failure event E3 is at the output of a PRIORITY-AND with the order specified as E4 before S, then we only have the path $A_2^i \rightarrow A_2^1 \rightarrow E_3$ in Fig. 4 as the corresponding EFSM. This makes it possible to use EFSMs to model sequential dependencies as required by the PRIORITY-AND gate.

The distinction between inclusive and exclusive-OR gates can be ignored in the fault tree analysis when dealing with independent, low probability component failures (see [1], page VII-7). Therefore we do not introduce special approaches to differentiate them in our method. If a distinction is truly needed, additional guards and transitions can be introduced on the model.

Algorithm 1 presents a systematic method to construct EFSMs in a modular way from a given fault tree. Note that the algorithm includes modelling of two types of gates only, AND and OR. However, it can be extended to include other types of gates like INHIBIT and PRIORITY-AND as discussed above. In Algorithm 1, lines 9-18 describe the modelling of OR gates and lines 19-30 describe AND gates. The addition of guards on the transitions mentioned in lines 16 and 28 describe the use of EFSM variables in guard conditions as shown in Fig. 3b for the OR gate, and in Fig. 4 for the AND gate, respectively.

4.2 Verification

In software controlled complex systems, safety analysis plays a significant role in formulating the safety requirements for the subsequent system design. Establishing confidence in the fault tree analysis is typically done manually. This is a shortcoming as it is error prone and even intractable for large and complex systems. An automated analysis method is very beneficial in providing sufficient verification evidence for the safety analysis phase. In this section, we present how typical specifications are modelled and verified using nonblocking verification algorithms in Supremica.

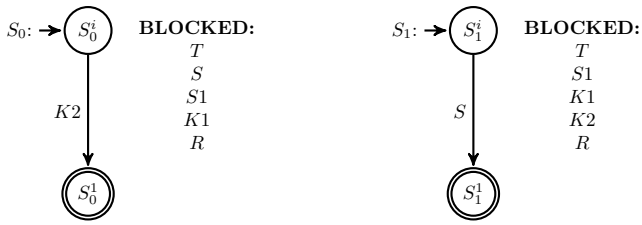
When system operation is started in the pressure tank in Fig. 1, the pump starts filling fluid into the tank. When the tank is full and the threshold pres-

Algorithm 1: Modular fault tree modelling

Input: Fault Tree, FT
Output: EFSM set corresponding to the fault tree, FT

```
Initialisation                                                          1
    declare basic events set, BE                                        2
    declare variables, Q, curr_node, child                              3
add root (FT) to Q // queue, Q contains elements to be processed        4
BE:= getBasicEvents (FT)                                                5
while Q ≠ ∅ do                                                          6
    curr_node:= pop (Q) // get the oldest element in queue              7
    gate:= getGate (curr_node) // retrieve connecting gate of node      8
    if gate is OR then                                                  9
        create initial and terminal locations, l_0 and l_n              10
        foreach child ∈ getChildren (gate) do                           11
            if child ∈ BE then // child is a basic event                12
                addTransition(l_0, l_n, child)                          13
            else // child is an intermediate event                      14
                addTransition(l_0, l_n, child)                          15
                add guards using automaton variables on the respective  16
                  transitions
                add child to Q                                          17
        markLocations(curr_node, root (FT))                             18
    else // node is an AND gate                                         19
        create initial and terminal locations, l_0 and l_n              20
        children:= getChildren (gate)                                   21
        create a set of strings, 𝕊, by permutation over children        22
        // each string is a path from l_0 to l_n
        foreach string ∈ 𝕊 do                                           23
            create transitions and locations correspondingly            24
            obtain the set of events, 𝔼                                 25
            foreach event ∈ 𝔼 do                                        26
                if event ∉ BE then // it is intermediate event          27
                    add guards using automaton variables on respective  28
                      transitions
                    add event to Q                                      29
        markLocations(curr_node, root (FT))                             30
function markLocations(curr_node, root (FT))                            31
    if curr_node == root (FT) then                                      32
        mark the terminal location, l_n                                 33
    else                                                                34
        mark all locations                                              35
function addTransition(l_a,l_b,event)                                   36
    add transition between l_a and l_b                                  37
    label transition with event                                         38
```

sure is reached, pressure switch S opens, causing K2 to open, and consequently the pump to stop. K2 failing to open would result in continuous pumping beyond the threshold and may result in the rupture of the tank. Therefore K2 is critical for safe operation and a primary failure of K2 may result in the top level failure event E1. Ideally, this behaviour should be captured in our FTA and we can verify this. Fig. 6a shows the EFSM modelling this specification. $K2$ is the only active event in this EFSM and the other basic events in the fault tree are blocked. Recall that transitions labelled by blocked events are removed in the synchronous composition of the specification and the plant models. Therefore, by blocking all basic events but $K2$, we ensure that $K2$ is included in the marked language of the EFSM whereas other basic events are not. A nonblocking verification performed on the synchronised model of this specification together with the plant models, shows that the system is nonblocking, thereby verifying that a primary failure of K2 is sufficient to cause rupture of the tank, the failure event E1.



(a) EFSM for $K2 \rightarrow E1$ **(b)** EFSM for $S \rightarrow E1$

Figure 6: EFSM for specifications

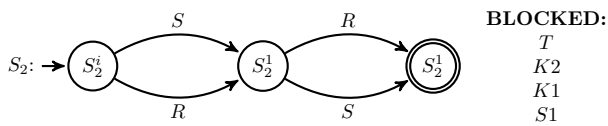


Figure 7: EFSM for specification $S \wedge R \rightarrow E1$

On the other hand, since we have the timer relay as a backup in the system, only a failure of the pressure switch, S, should not lead to tank rupture. We can model this as a specification shown in Fig. 6b. Since we are only interested

in the primary failure of pressure switch S, we block the remaining basic events in the fault tree. A nonblocking verification of this specification synchronised with the plant model results in a blocking state, thereby verifying that only S occurring will not result in the top level failure event E1. However, if we also include the failure of the timer relay R, we get the specification as shown in Fig. 7. With this specification, we can verify that the system is indeed nonblocking, i.e., a failure of both components S and R will lead to the top level failure event E1. Specifications to model the remaining causes leading to the top level event and/or the intermediate events are done in a similar way as in figures 6 and 7.

The type of specifications that we have seen so far are modelled to check whether certain basic events or combinations of events lead to a failure event. Given such a specification, SP, and fault tree, FT, Algorithm 2 presents how EFSM models can be obtained from them.

Algorithm 2: Modelling specifications

Input: Fault Tree, FT and Specification, SP
Output: EFSM modelling the specification
Initialisation 1
  declare basic events set, BE 2
  declare active events set, AE 3
  declare blocked events set, BLOCKED 4
BE:= `getBasicEvents` (FT) 5
AE:= `getBasicEvents` (SP) 6
create locations $l_0, l_1, ..., l_N$ with $N = |\mathsf{AE}|$ 7
make l_0 the initial location 8
make l_N the single marked location 9
for every pair (l_{i-1}, l_i) with $i \in \{1, 2, ..., N\}$ create N transitions 10
label each transition uniquely from $\sigma \in$ AE 11
add blocked events, BLOCKED:= BE $\setminus$ AE 12

4.3 Minimal Cut Sets

Our approach is not only useful for verification but also in calculating minimal cut sets, one of the most prominent qualitative analysis techniques of standard fault trees. A *cut set* is a set of component failure events that together lead to the top level failure. Formally, a minimal cut set is a *smallest* combination of

component failures which, if they all occur, lead to the top level failure event. It is smallest in the sense that all failures are needed for the top level event to occur and if one of them in a cut set does not occur, then the top event will not occur by that set. For example, the minimal cut sets for the pressure tank system are {T}, {K2}, {S, S1}, {S, K1}, {S, R}.

In our modelling approach presented in Section 4.1, the marked locations in the composed model correspond to the top level failure event. This makes it possible to use the marked language of the plant EFSM to calculate the minimal cut sets. In our case, a cut set is a set of events that lead to marked locations corresponding to the top level failure event. Calculating minimal cut sets is then done by finding the shortest paths in the synchronised plant EFSM from the initial location to the marked locations, a task typically solved by variants of breadth-first search algorithms. Algorithm 3 presents one such method to calculate minimal cut sets by exploiting the marked language of the synchronised EFSM. Lines 11-13 of the algorithm adds the basic events that can reach the marked location in the synchronised EFSM to the output set. Lines 14 and 15 ensure that the same events are not repeated.

5 Conclusion

We have shown how fault tree analysis can be formalised to be automatically analysed by modelling techniques from Supervisory Control Theory (SCT) using the tool Supremica. We present a systematic approach to incrementally obtain formal models from a given standard fault tree, as summarised in Algorithm 1. Algorithm 2 describes a method to automatically generate specifications for given properties of the fault tree, so that theycan be verified using non-blocking verification. Finally, Algorithm 3 presented a method to automatically calculate minimal cut sets from the generated models.

Though our modelling approach can model complex systems with redundant architectures and dynamic dependencies, we here limit ourselves to the standard symbols described in the *Fault Tree Handbook*. Our approach can indeed be extended to use *dynamic* gates. The formal model obtained from the approach discussed in this paper, considers only the fault behaviour of the system as described by a given fault tree and nothing else. While we verify certain properties on the model to establish confidence in the system, we do not focus on correctness of the construction of the fault tree in the context of

Algorithm 3: Computation of Minimal Cut Sets

Input: $\mathsf{EFSM}_1, \ldots, \mathsf{EFSM}_n$ modelling the considered FT
Output: Set of minimal cut sets, S
Initialisation 1
  declare variable Q as queue with states to be processed 2
  declare synchronised EFSM A as $\mathsf{EFSM}_1 \parallel \ldots \parallel \mathsf{EFSM}_n$ 3
  declare basic event set, BE 4
  declare blocked events set, BLOCKED 5
$\mathsf{BE} := \texttt{getBasicEvents}(\mathsf{A})$ 6
while $\exists e \in \{\sigma \mid \exists s' \ s.t. \ (s_i, \sigma, s') \in \rightarrow_A \wedge \ \sigma \in \mathsf{BE}\}$ **do** 7
  $\mathsf{Q}.\texttt{put}(s_i)$ // Enqueue the initial state s_i 8
  while $\mathsf{Q} \neq \varnothing$ **do** 9
    $s := \mathsf{Q}.\texttt{get}()$ // Dequeue state s from Q 10
    if $\exists s', \exists \sigma \ s.t. \ (s, \sigma, s') \in \rightarrow_A \wedge \ \texttt{isMarked}(s')$ **then** 11
      // Retrieve basic events labelling transitions from s_i to s
      $\Sigma_c := \texttt{getEvents}(s_i, s') \ \cap \ \mathsf{BE}$ 12
      // Σ_c is one minimal cut set, insert it into S
      $\mathsf{S}.\texttt{put}(\Sigma_c)$ 13
      create a single location (marked) EFSM_{sp} with BLOCKED $:= \Sigma_c$ 14
      // Update A by blocking all basic events in Σ_c
      $\mathsf{A} := \mathsf{A} \parallel \mathsf{EFSM}_{sp}$ 15
      break 16
    else 17
      forall $s' \ s.t. \ (s, \sigma, s') \in \rightarrow_A$ **do** $\mathsf{Q}.\texttt{put}(s')$ 18

the system's operational environment. In a behavioural approach, we would formally model the complete behaviour of the system, i.e., including the nominal operational behaviour and not only the fault behaviour. This presents a wide range of possibilities. One possible extension is to adopt a formal approach similar to model checking [5]. Another notable extension of our work is to use the behavioural system models and the supervisor synthesis framework provided by SCT to automatically synthesize the fault behaviour. This falls in line with the model-based dependability analysis [10] approach for safety analysis. In such extensions, the system model becomes the plant models and the work in this paper can then be used to obtain formal specifications from a given fault tree. This approach makes it possible to use such formal models in several stages of a model-based design process. The state based models that are created can be re-used during the development of the software programs

in the later stages. The work presented in this paper can provide a solid basis for possible extensions in those areas.

A primary motivation for this work is our current focus on formal verification of autonomous driving systems where SCT and Supremica have been used to verify software for autonomous driving systems [17]. We believe our work in this paper will strongly encourage the application of SCT and Supremica in different stages of safety critical software development starting from safety analysis in the early stages to synthesis and verification of the software in the end stages. Our work in this paper is successfully integrated with a model-based systems engineering tool [13], that is widely used in the automotive industry.

References

[1] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook", Nuclear Regulatory Commission Washington DC, Tech. Rep., 1981.

[2] S. Kabir, "An overview of fault tree analysis and its application in model based dependability analysis", *Expert Systems with Applications*, vol. 77, pp. 114–135, 2017.

[3] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools", *Computer science review*, vol. 15, pp. 29–62, 2015.

[4] K. M. Hansen, A. P. Ravn, and V. Stavridou, "From safety analysis to software requirements", *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 573–584, 1998.

[5] A. Thums and G. Schellhorn, "Model checking FTA", in *International Symposium of Formal Methods Europe*, Springer, 2003, pp. 739–757.

[6] J. Xiang, K. Ogata, and K. Futatsugi, "Formal fault tree analysis of state transition systems", in *Fifth International Conference on Quality Software (QSIC'05)*, IEEE, 2005, pp. 124–131.

[7] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems", *IEEE Transactions on reliability*, vol. 41, no. 3, pp. 363–377, 1992.

[8] B. Kaiser, C. Gramlich, and M. Förster, "State/event fault trees—a safety analysis model for software-controlled systems", *Reliability Engineering & System Safety*, vol. 92, no. 11, pp. 1521–1537, 2007.

[9] G. K. Palshikar, "Temporal fault trees", *Information and Software Technology*, vol. 44, no. 3, pp. 137–150, 2002.

[10] S. Sharvia, S. Kabir, M. Walker, and Y. Papadopoulos, "Model-based dependability analysis: State-of-the-art, challenges, and future outlook", in *Software Quality Assurance*, Elsevier, 2016, pp. 251–278.

[11] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes", *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[12] R. Malik, "Programming a fast explicit conflict checker", in *2016 13th International Workshop on Discrete Event Systems (WODES)*, IEEE, 2016, pp. 438–443.

[13] SYSTEMITE, *Systemweaver*, `https://www.systemweaver.se/`, Accessed: 2020-05-09.

[14] M. Skoldstam, K. Akesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables", in *2007 46th IEEE Conference on Decision and Control*, IEEE, 2007, pp. 3387–3392.

[15] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional nonblocking verification of extended finite-state machines", *Discrete Event Dynamic Systems*, vol. 26, no. 1, pp. 33–84, 2016.

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

[17] Y. Selvaraj, W. Ahrendt, and M. Fabian, "Verification of decision making software in an autonomous vehicle: An industrial case study", in *Formal Methods for Industrial Critical Systems*, K. G. Larsen and T. Willemse, Eds., Cham: Springer International Publishing, 2019, pp. 143–159, ISBN: 978-3-030-27008-7.

PAPER C

Automatically Learning Formal Models: An Industrial Case from Autonomous Driving Development

Yuvaraj Selvaraj, Ashfaq Farooqui, Ghazaleh Panahandeh, Martin Fabian

ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada,
 DOI: 10.1145/3417990.3421262

The layout has been revised.

Abstract

The correctness of autonomous driving software is of utmost importance as incorrect behaviour may have catastrophic consequences. Though formal model-based engineering techniques can help guarantee correctness, challenges exist in widespread industrial adoption. One among them is the model construction problem. Manual construction of formal models is expensive, error-prone, and intractable for large systems. Automating model construction would be a great enabler for the use of formal methods to guarantee software correctness and thereby for safe deployment of autonomous vehicles. Such automated techniques can be beneficial in software design, re-engineering, and reverse engineering. In this industrial case study, we apply active learning techniques to obtain formal models from an existing autonomous driving software (in development) implemented in MATLAB. We demonstrate the feasibility of active automata learning algorithms for automotive industrial use. Furthermore, we discuss the practical challenges in applying automata learning and possible directions for integrating automata learning into automotive software development workflow.

1 Introduction

In recent years, the global automotive industry has made significant progress towards the development of autonomous vehicles. Such vehicles potentially have several benefits including the reduction of traffic accidents and increased traffic safety [1]. However, these are highly complex and safety critical systems, for which correct behaviour is paramount, as incorrect behaviour can have catastrophic consequences. Ensuring safety of autonomous vehicles is a multi-disciplinary challenge, where software design and development processes play a crucial role. A strong emphasis is placed on updating current engineering practices to create an end-to-end design, verification, and validation process that integrates all safety concerns into a unified approach [2].

Automotive software engineering is faced with several challenges that in-

clude non-technical aspects (such as organization, strategic processes, etc.) and technical aspects (such as the need for new methodologies that combine traditional control theory and discrete event systems, quality assurance for reliability, etc.) [3], [4]. Model-based engineering techniques can address some of the challenges and help tackle the complexity in developing dependable automotive software [5]–[7]. An autonomous vehicle consists of several software and hardware components that interact to solve different tasks. Software in a modern car typically consists of hundreds of thousands of lines of code deployed over several distributed units developed by different suppliers and OEMs. The model-based approach to design, test, and integrate software systems using sufficiently detailed black-box and white-box models is instrumental in achieving the necessary correctness guarantees for such complex systems. In this vein, several tools and methods have been developed over the years and model-based design with MATLAB/Simulink has become increasingly successful in a number of automotive companies [8].

A direct consequence of such increasing software complexity is the possible presence of potentially dangerous edge cases, bugs due to subtle interactions, errors in software design and/or implementation. Although the automotive industry is constantly evolving, testing (including model-based testing) is currently a prominent technique for software quality assurance [9]. However, an approach only based on testing is insufficient and partly infeasible to guarantee the correctness of autonomous vehicles [10]. Thus, there is a need for strict measures for quality assurance and the use of formal methods in this regard promise to be beneficial [11], [12].

Formal verification techniques can indeed be used to identify design errors in Simulink models using Simulink Design Verifier (SDV) and also to perform static code analysis on generated C code using Polyspace [13]. However, there are limitations in SDV; for instance in scalability and in the verification of temporal properties that cannot be expressed as assertions [14], [15]. SDV also requires the models to be built in Simulink. This presents a challenge in reasoning about MATLAB code without Simulink function blocks. Also, SDV and Polyspace cannot be used to reason about black-box models which might be necessary to guarantee the correctness of the complete system under design. Thus, in such cases there is a need to use complementary methods.

In [16], different formal verification methods were used to verify an existing decision making software (developed using MATLAB) in an autonomous

driving vehicle. Formal models of the code were manually constructed to perform formal verification and several insights were presented. Admittedly, formal methods can be relatively more beneficial if introduced during the early stages of the software development workflow rather than being used for post-hoc verification after development. However, there are several obstacles that impede the widespread adoption of formal methods [11], [17]. Significant trade-offs (e.g. tools compatible with formal methods, development cost and time, etc.) have to be made that disrupts current industrial best practice. Therefore, any work towards industrial adoption of formal methods in the automotive domain without significant disruptions on current practice is definitely rewarding.

Formal verification techniques like model checking [18]—to prove the absence of errors in software designs—or, formal synthesis techniques like supervisor synthesis [19]—to generate a controller/supervisor that is correct by construction—require a model that describes the behaviour of the system. However, constructing a formal model that captures the behaviour of the software under design is a challenging task and is one of several impediments in the industrial adoption of formal methods. Manual construction of models is expensive, prone to human errors, and even intractable for large systems. Constructing such a model manually is also time consuming, which further complicates things as the implementation typically changes frequently, especially so for rapidly evolving systems like autonomous driving.

Automating model construction could help to speed up industrial adoption of formal methods by reducing the burden of manually constructing the models. Such automated methods will significantly help find potential errors as they can automatically generate and verify production code at regular intervals. This will further strengthen the suitability of formal methods for industrial deployment [17], [20]. Automatically constructing formal models can also help understand and reason about ill-documented legacy systems and *black-box* systems which is crucial for the quality assurance in large-scale and complex automotive systems.

Active automata learning [21]–[28] is an active field of research that addresses the problem of automatic model construction. These approaches constitute a class of machine learning algorithms that aim to deduce a finite-state automaton describing the behaviour of a system by actively interacting with the target system. In this paper, we apply active automata learning to ob-

tain formal models from existing autonomous driving software under development in MATLAB/Simulink. To this end, we adapt a state exploration based algorithm [26] and a minimal language learning algorithm [21], to automatically construct behavioural models of the software. We present results from the learning outcomes and discuss practical challenges in the process. Note that this case study does not aim to compare the performance of the two algorithms, but to show the applicability of active automata learning in a MATLAB/Simulink development environment.

Model validation was done in several ways, though not formally, that strengthen the confidence in the usefulness of the automatic model construction. Simulating with the same input parameters both the actual MATLAB code, and the learnt model resulted in similar outputs, even to the extent that a known bug existing in the actual MATLAB code was also present in the learnt model. Visually comparing the learnt model to a manually constructed model of the same development code [16] showed obvious similarities. Furthermore, language minimisation of the learnt model resulted in a model that matches the abstract model created manually during the early stages of the software design (Figure 2).

The paper is structured as follows. Section 1.1 presents a brief overview of related work and Section 2 presents the necessary preliminaries. In Section 3, we briefly describe the system under learning (SUL) followed by a description of the learning framework and the validation of the formal model learned in Section 4. Section 5 presents our insights from this case study, where we discuss practical challenges and possible directions for integrating automata learning into automotive software development workflow. The paper is concluded in Section 6.

1.1 Related Work

Automatically extracting finite-state models for formal verification has been done previously, for instance, from Java source code in [29], and from C code in [30], [31]. These methods rely on extracting an automaton by parsing the program source code. Hence, they are language specific and strictly rely on well defined coding patterns and program annotations. Also, it is not possible to extract models where the source code is not available, such as black-box systems. Active automata learning mitigates these restrictions and learns models of black-box systems through interaction.

There exist work on integrating MATLAB/Simulink development environment with tools compatible for formal verification. For example, in [32], MATLAB/Simulink models are translated to an intermediate language that can later enable the use of SMT solvers for verification. Other works include developing MATLAB toolboxes to integrate with a theorem prover [33] and a hybrid model checker [34]. Such methods depend on considerable manual (and skilled) work to understand the semantics of the MATLAB/Simulink models and to develop the respective toolboxes. In contrast, the work presented in this paper removes such dependencies and learns the formal model by actively interacting with the MATLAB code.

Active automata learning has been successfully applied to learn and verify communication protocols using Mealy machines [22], [35], and to obtain formal models of biometric passports [36] and bank cards [37]. In [38], automata learning is used to learn embedded software programs of printers. Though such research indicates the use of active automata learning for real-life systems, challenges exist to broaden its impact for practical use [25], [39], [40]. There are very limited examples on the use of active automata learning in an automotive context [41], [42] and it is yet to find its place in automotive software development. To the best of our knowledge, active automata learning has not been used previously to learn behavioural models from automotive software implemented in MATLAB/Simulink, a common model-based development tool within the automotive industry.

2 Preliminaries

An *alphabet*, denoted by Σ, is a finite, nonempty set of events. A *string* is a finite sequence of events chosen from the alphabet. The empty string, denoted by ε is the string with zero events. For two strings, s and t their *concatenation* is denoted by st, i.e., the string formed such that s is followed by t. The set of all strings of certain length, k from an alphabet, Σ is denoted as Σ^k. Thus, Σ^2 is defined as $\Sigma\Sigma$ and similarly, $\Sigma^{(n+1)} = \Sigma^n\Sigma$. The set of all strings of finite length over an alphabet Σ, including $\Sigma^0 = \{\varepsilon\}$, is denoted by Σ^*.

A *language* $\mathcal{L} \subseteq \Sigma^*$ is a set of strings over Σ. A string s is a *prefix* of a string u, if there exists a string t such that $u = st$. For a string $s \in \mathcal{L} \subseteq \Sigma^*$, its *prefix-closure* $\overline{s}$ is the set of all prefixes of s, including s itself and ε. $\mathcal{L}$ is said to be *prefix-closed* if the prefix-closures of all its strings are also in $\mathcal{L}$,

that is $\overline{\mathcal{L}} = \mathcal{L}$. Suffix-closure can be defined analogously.

Definition 1 (State)**:** *Let* $V = \{v_1, v_2, ..., v_n\}$ *be a set of* state variables*, where each variable* v_i *has a discrete finite domain defined as* v_i^D*. A state* q *is then defined as the assignment of values to variables,* $\hat{V} \in V^D$ *where* $V^D = v_1^D \times v_2^D \times \cdots \times v_n^D$*. We call* $\hat{V}$ *a* valuation*.*

Definition 2 (DFA)**:** *A* deterministic finite automaton *is defined as a 5-tuple* $\langle Q, \Sigma, \delta, q_0, M \rangle$*, where:*

- Q *is the finite set of* states*;*
- Σ *is the* alphabet *containing the events;*
- $\delta : Q \times \Sigma \to Q$ *is the* partial transition function *that takes a state and an event as arguments and returns a state;*
- $q_0 \in Q$ *is the* initial state*;*
- $M \subseteq Q$ *is the set of* marked states*.*

The set of all deterministic finite automata is denoted $\mathcal{A}$. Given $A \in \mathcal{A}$, we have the notion of language *generated* and language *marked* by the automaton [43]. The language generated by the automaton, $\mathcal{L}(A)$ is the set of all strings defined from its initial state, q_0. The marked language, $\mathcal{L}_m(A) \subseteq \mathcal{L}(A)$ is the set of all strings that reach marked states. While the generated language denotes behaviour that is possible but not necessarily accepted, the marked language denotes behaviour that is accepted. A language is said to be *regular* if it is marked by some DFA. It is well-known [43] that for a given regular language, there exists a *minimal* automaton, in the sense of least number of states and transitions, that accepts that language.

2.1 The L^* Algorithm

The L^* algorithm [21] is a prominent algorithm in the field of active automata learning, that has inspired a tremendous amount of work yielding positive results. It learns a minimal automaton $\hat{\mathcal{M}}$ accepting a marked regular language $\mathcal{L}_m(\hat{\mathcal{M}}) \subseteq \Sigma^*$, that represents the behaviour of the SUL, over a finite alphabet Σ. The L^* algorithm assumes access to an *oracle* that has complete knowledge of the system, and the algorithm works by posing *queries* that are answered by the oracle. However, to practically use L^*, a mechanism to implement the

different queries needs to be put in place. In this paper, we use a modified version of the L^* described in [44], [45]. The learning algorithm interacts with the SUL using two types of queries:

Membership Queries: Given a string $s \in \Sigma^*$, a membership query for s returns 2 if the string can be executed by the SUL and takes the system (from the initial state) to a marked state. If the string can be executed but does not reach a marked state, 1 is returned. Else, 0 is returned. The membership query has the signature: $T : \mathcal{A} \times \Sigma^* \rightarrow \{0, 1, 2\}$, and for $A \in \mathcal{A}$ and $s \in \Sigma^*$ we have:

$$T(A, s) = \begin{cases} 2, & s \in \mathcal{L}_m(A) \\ 1, & s \in \mathcal{L}(A) - \mathcal{L}_m(A) \\ 0, & \text{otherwise.} \end{cases} \tag{C.1}$$

Equivalence Queries: Given a *hypothesis* automaton $\mathcal{H}$, an algorithm verifies if $\mathcal{H}$ accurately represents the language $\mathcal{L}_m(\hat{\mathcal{M}})$. If not, a *counterexample* $c \in \Sigma^*$ must be provided, such that, c is incorrectly accepted or incorrectly rejected by $\mathcal{H}$.

In [21], a probabilistic method to generate counterexamples is proposed by performing a random walk on the hypothesis. Other methods to generate counterexamples could include complete exploration of the hypothesis state-space, or by borrowing ideas from the testing community, such as the W-method [46], or the Wp-method [47]. In this paper, we use the W-method.

Let $\hat{\mathcal{M}}$ have n states. Given a hypothesis $\mathcal{H}$, with m states, the W-method creates test strings to iteratively extend the hypothesis containing m states until it has $n \geq m$ states. To do so, the W-method uses two sets, P and W, where P contains strings that reach some state in $\mathcal{H}$, at least one for each state in $\mathcal{H}$. The set W contains strings such that every pair of states reached by strings in P can be distinguished based on where they reach when continued with some strings in W.

The W-method generates test strings according to PUW where $U = (\Sigma^0 \cup \Sigma \cup \Sigma^2 \cup ....\Sigma^{n-m})$. The generated test strings are executed in the hypothesis as well as the actual system to find any inconsistencies. To test a string s in the actual system, it suffices to perform a membership query on the string, while testing the string in the hypothesis is done as $T(\mathcal{H}, s)$. If a mismatch between the two is found, that string s is a counterexample, the testing is terminated, and the counterexample is sent to the learner. This is repeated until $n = m$ or all the generated test strings have been tested.

At any given time, the learner updates its knowledge about the target language. Internally, this knowledge is represented as an *observation table*. The observation table is a two dimensional table with the rows indexed by prefix-closed strings, and the columns indexed by suffix-closed strings. The cells contain the value according to the membership query for the concatenated string in the corresponding row and column.

The learner, by using membership queries, populates the observation table such that the table is closed and consistent as defined in [44], [45]. Then, the learner performs an equivalence query on the hypothesis automaton constructed from a closed and consistent observation table. If a counterexample is found, it and all its prefixes are added to the observation table. This process repeats until no counterexample can be found.

2.2 The Modular Plant Learner

The Modular Plant Learner (MPL) algorithm [26] is a state based active learning algorithm specifically developed to learn a *modular* model, that is, one composed of a set of interacting automata, *modules* that together define the behaviour of the system. It does this by actively exploring the state space of a program in a breadth-first search manner provided with knowledge about the modules. These modules are defined using three pieces of information: a name for each module (m), the subset of events (Σ_m) that belong to the module, and the subset of state variables that either affect or are affected by events in the module. In most cyber-physical systems today, it is possible to observe the state variables. The MPL leverages on this possibility to observe the internal state variables of the SUL to learn a modular model in a smart way.

The MPL requires an interface by which it can affect and observe the SUL, the set of events, and the initial state of the system. The algorithm interacts with the system that is to be modelled, and actively queries it to learn what states are reachable from the initial state. Furthermore, some prior knowledge of the structure of the system is used to split the learning into several modules.

The algorithm consists of two components, the *Explorer* and the *ModuleBuilder*. When launched, the algorithm, starts the *Explorer* and one instance of the *ModuleBuilder* for each module defined. The *Explorer* is responsible for exploring the new states, and the *ModuleBuilder* keeps track of the module as it is learned.

The *Explorer* maintains a queue of states that need to be explored, terminating the algorithm when the queue is empty. The learning is initiated by adding an initial state to the queue, which becomes the starting state of the search. For each element in the queue, the *Explorer* checks if an event from the alphabet Σ can be executed. This is achieved using the interface to the system. If a transition is possible, the Explorer broadcasts the current state (q), the event (σ) and the state reached (q') to all the *ModuleBuilder*s.

The *ModuleBuilder* tracks the learning of each module as an automaton. This is done by maintaining a set Q_m containing the states of the module, and a transition function $T_m : Q_m \times \Sigma_m \rightarrow Q_m$, for each module. The *ModuleBuilder*, on receiving the broadcast, evaluates if the received transition is of interest to the particular module. If it is of interest, the transition is refined according to the prior knowledge of the modules, and added to the module; else it is discarded. When the valuation of state variables in the source state is equal to that in the reached state, the transition becomes a self-loop.

Once the transition is processed, the *ModuleBuilder* waits for further broadcasts from the *Explorer*. The algorithm terminates when all modules are waiting for broadcasts, and the exploration queue is empty. Each *ModuleBuilder* can now construct and return an automaton based on Q_m and T_m.

3 System Under Learning

In this case study, we focus on learning the behavioural model of the *Lateral State Manager* (*LSM*), a sub-component of the decision making and planning module in an autonomous driving system. The software for this module is obtained from Zenuity, one of the leading companies in the development of safe and reliable software for autonomous driving software and advanced driver assistance systems. The system under learning (SUL), the *LSM*, is responsible for managing modes during an autonomous lane change. The lane change software module is implemented in object-oriented MATLAB-code [48] using several classes with different responsibilities. Simulations during the development of *LSM* code are made using the MATLAB/Simulink environment. A simplified overview of the system and the interaction of *LSM* with a high level strategic planner (*Planner*) and a low level planner (*Path Planner*) is shown in Figure 1. The lane change module is cyclically updated at a high frequency

with the current vehicle state, surrounding traffic state, and other reference signals.

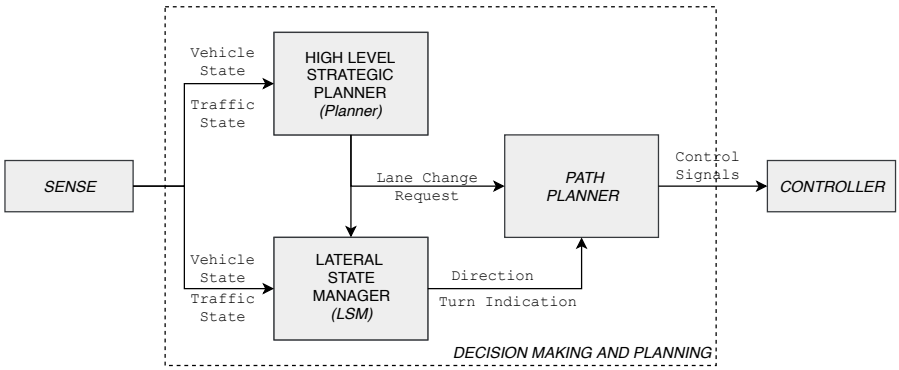


Figure 1: The lane change module system overview and interactions between the three components: *Planner*, *LSM*, and *Path Planner*.

The *Planner* in the lane change module is responsible for strategic decisions. Depending on the state of the vehicle, *Planner* sends lane change requests to *LSM*, indicating the desired lane to drive in. This request is sent in the form of a `laneChangeRequest` signal, which takes one of the three values: `noRequest`, `changeLeft`, or `changeRight` at any point in time. On receiving a request, *LSM* keeps track of the lane change process by managing the different modes possible during the process, and issues commands to the *Path Planner*. If a lane change is requested, the *Path Planner* plans a path and sends required control signals to the low level controller to perform a safe and efficient lane change. Due to the nature of the task to solve, the *LSM* implements a finite state machine. A meta-level abstraction of the *LSM*, which consists of seven states, is shown in Figure 2. For confidentiality reasons, the state and event names are not detailed. An example of a state in the *LSM* state machine is *State_Finished* that represents the completion of the lane change process.

The implementation of *LSM* is done with a set of methods and variables, one of which is the current state variable. A call to *LSM* is issued at every update cycle. During each call, the *LSM* undergoes three distinct execution stages, where the respective set of methods are executed. In the first stage, all the inputs are updated according to the function call arguments. In this stage, an associated function called `updateState` is executed which is responsible for

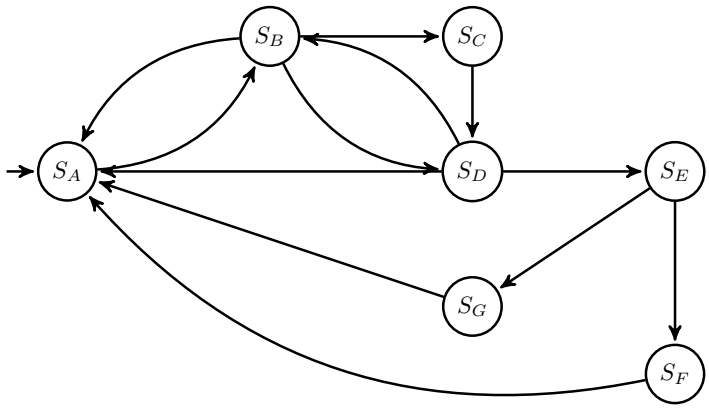


Figure 2: A meta-level finite-state abstraction of the *LSM*. The state names and events are not detailed due to confidentiality reasons.

passing the control flow to the next stage. Second, depending on the current state, code is executed to decide whether the system transits to a new state or not. This code also assigns outputs and internal variables. Finally, if a transition is performed, the last stage executes code corresponding to the new state entered and assigns new values to the variables.

4 Learning setup

To actively learn a DFA model of the SUL, which in our case is the *LSM*, an interface is necessary through which it is possible to execute (strings of) events. These event strings represent the executable actions on the SUL. The execution of these actions results in changing the program state of the *LSM*. If an event is requested that is not executable by the SUL in the current state, the SUL should reply with an error message. Also, it should be possible to observe and set the state of the SUL. Figure 3 presents an overview of the active automata learning setup used in this case study. This consists of three components: the *system under learning*, the *learner*, and the *interface* between the SUL and the learner. In addition to these, Figure 3 also includes the *formal model analyser*, that is used to validate the learnt models. The learner consists of Scala [49] implementations of the two learning algorithms

described in Section 2. The learning setup allows learning of automata models by (actively) interacting with the SUL. The following subsections describe the components in brief detail, the learning outcome and the steps taken to validate the learnt model of the *LSM*.

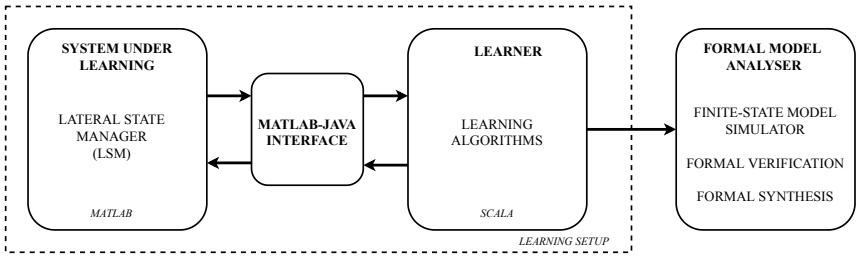


Figure 3: Overview of the learning setup.

4.1 Abstracting the Code

As described in Section 3, the *LSM* is a part of the lane change module, which is updated cyclically at a high frequency with the necessary signals. A call to the *LSM* is issued at every execution cycle. Recall that the *LSM* is implemented using a set of methods and during each call the *LSM* undergoes three execution stages, where the respective set of methods are executed. In order to decide whether the system transits to a new state or not, the *LSM* is dependent on external function calls. These interactions with external modules need to be abstracted away to learn a model of the *LSM*. Thus, the first step in the learning process is to abstract the MATLAB code such that all external dependencies are removed. The abstraction process is described using a small example as follows.

Example 4.1

Consider the small code snippet shown in Listing 9.1. It consists of a function `duringStateA` that takes input arguments and depending on the variables `var1` and `var2` decides whether the system transits to a new state or not. The values of these two variables are dependent on external function calls, `function1` and `function2`, respectively.

Listing 9.1: A small illustrative example

```
function duringStateA(self,
                      laneChangeRequest)
    var1 = function1();
    var2 = function2(laneChangeRequest);
    if var1 && var2
        self.state = stateB;
    end
end
```

These external function calls can be abstracted away by replacing each of the external calls with additional parameter variables that are passed as input arguments to `duringStateA`. This is shown in Listing 9.2 using the added input argument `decisionVar`. The two variables, `decisionVar.var1` and `decisionVar.var2`, have the domain $\{True, False\}$. While this abstraction increases the number of input parameters to the function, the decision logic remains unchanged.

Listing 9.2: Abstracted version of Listing 9.1

```
function duringStateA(self,
                      laneChangeRequest,
                      decisionVar)
    if decisionVar.var1 && decisionVar.var2
        self.state = stateB;
    end
end
```

Similarly, all such external function calls are abstracted and the final abstracted function contains one additional parameter, `decisionVar` as an input argument to the `updateState` function. The output of the `updateState` function is a set of internal variables which includes the current state and the direction for the lane change among others. This set of variables are used by the learner to observe the behaviour of the *LSM* during their interaction, which is described in detail in the following section.

4.2 Interaction With the SUL

In active learning, the learner actively interacts with the SUL and reasons on the observed behaviour to construct a model of the SUL. Hence, this interaction is crucial between the learning algorithms implemented in Scala and the *LSM* implemented in MATLAB. In order to facilitate this interaction, we need to:

1. create an interface between the learner and the SUL,

2. provide information to the learner on how to execute the *LSM* and observe the output.

The learner must be able to call MATLAB functions, evaluate MATLAB statements, pass data to, and get data from MATLAB. Scala source code is compiled to Java bytecode and the resulting executable code is run on a Java virtual machine (JVM). Therefore, the interface essentially integrates Java with the MATLAB environment using the MATLAB Engine API for Java [50].

With this interface established, the learner can now call the `updateState` function by providing an input assignment to the state variables and the input variables `laneChangeRequest` and `decisionVar`. However, to learn a DFA model from a given SUL, the learner additionally requires, among other things, predicates over state valuations that define the marked states, the set of events, and event predicates that define when an event is enabled or disabled.

Since the interaction between the learner and the *LSM* is done by the `updateState` function, the input parameters are used to define the alphabet – the set of events that are executable by the *LSM*– of the model. Each unique valuation of these input parameters corresponds to an event in the alphabet. Since the abstracted *LSM* module is provided to the learning program, each function that is abstracted into a decision variable, potentially, results in one additional input parameter. Following the abstraction described in Section 4.1, ten external function calls in the *LSM* were abstracted to have ten Boolean valued `decisionVar`, in addition to one three valued `laneChangeRequest`, as input parameters. This results in a total of 3072 events. However, as unique mode changes in the *LSM* is defined only for a subset of these events, some of them would potentially not have any

effect on the model behaviour, and therefore their event predicates would be unsatisfiable.

The event predicates are defined over the state variables. The granularity of these predicates contribute to the performance of the learning algorithm. A very detailed predicate will potentially reduce the total number of strings to test in the SUL. A general rule of thumb for constructing these is to create one predicate for each abstracted variable. Taking the example from the previous section, all events corresponding to `decisionVar.var1` and `decisionVar.var2` are enabled when the predicate, `self.state == stateA` evaluates to *True*. For an event to be enabled in a given state, all individual predicates corresponding to the different variables must evaluate to *True*. Events with unsatisfiable predicates can be discarded. Doing so for the *LSM* results in a total of 1536 events.

Finally, to observe the behaviour of the *LSM*, the learner requires a set of variables. This is given by the output of the `updateState` function, which is the valuation of the internal variables, one of which is the current state variable that can at any point take the value of one of the seven states shown in Figure 2. Furthermore, the initial state (initial valuation of the variables) of the *LSM* is known and is provided to the learner.

4.3 Learning Outcome

Given an interface to the SUL and its alphabet, the learning algorithms described in Section 2 are applied to learn a model of the SUL. The algorithms were run on an Intel i7 machine, with 8GB ram, running Linux. This section discusses the outcome of the learning. Note that our aim was not to benchmark L^* and MPL against each other, but to show the applicability of active automata learning in an engineering tool chain based on MATLAB/Simulink.

Learning with L^*

The implemented L^* algorithm was unable to learn a complete model without running out of memory in our experiments. After 7 hours of learning, it was observed that 5 iterations of the hypothesis involved $500k$ membership queries and resulted in a hypothesis model with 8 states and 138 transitions. No insights could be drawn from the obtained partial model.

Two main obstacles were faced while learning using the L^*. Firstly, the

growth of the observation table. As the size of the table grows, it takes longer to check and ensure that the table is closed and consistent. Furthermore, the memory used to store the table grows rapidly by a factor dependent on the size of the alphabet. Secondly, exhaustive search for a counterexample using the W-method in the given setup is time consuming. Due to the large alphabet size the test strings result in long sequences which in-turn slow down the equivalence queries since these are implemented as multiple membership queries.

Learning with MPL

Apart from the interface with the SUL the MPL requires information about the modules to learn from the SUL. The *LSM* is a monolithic system and cannot, in its current form, be divided into modules. Hence, the MPL, though specifically developed to learn a modular system consisting of several interacting automata, learns a monolithic model of the SUL.

The resulting automaton consists of 37 states and 687 transitions. The learning took a total of 68 seconds. Furthermore, applying language minimisation [43] to the learnt model results in a model with 6 states and 114 transitions. The language minimised automation is shown in Figure 4, and its similarity to Figure 2 is obvious. The two states S_G and S_F of Figure 2 are bisimilar [43], so they both correspond to the single state q_6 of Figure 4.

The self-loops in the states of Figure 4 correspond to those events that are enabled in that particular state but do not change the internal state of the *LSM*. For example, consider the code snippet in Listing 9.2. When `decisionVar.var1` is *True* and `decisionVar.var2` is *False*, the corresponding event is enabled in `stateA`, but when fired does not cause a change in the value of `self.state`, and thereby results in a self-loop. Similarly, all such enabled events that do not change the internal state become self-loops in the learnt model. The state q_6 does not have a self-loop as it is a transient state in the *LSM*. That is, irrespective of input parameters, when q_6 is reached, *LSM* transits to state q_0 for every enabled event.

4.4 Model Validation

As described in Section 4.3, a formal model of the *LSM* is learnt using the MPL algorithm. In order to validate the learnt model, similar to [36], we com-

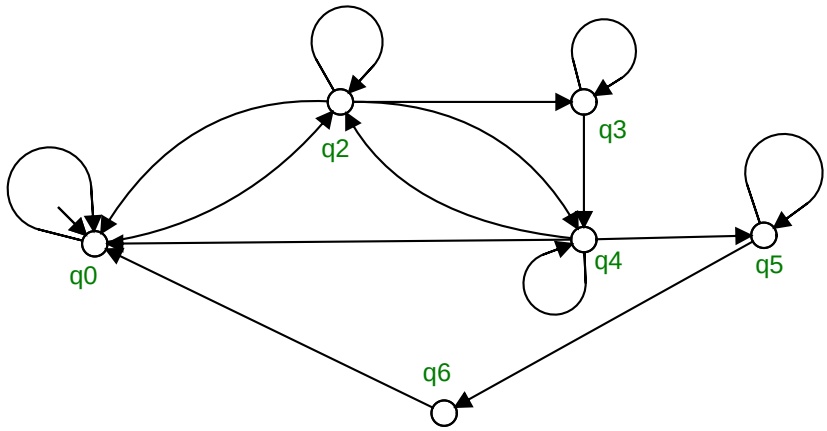


Figure 4: Learned model of the *LSM*. The state names and events are not detailed due to confidentiality reasons.

pared the learnt model to a model manually constructed from the MATLAB code. This was done using the tool SUPREMICA [51], which is an integrated development environment for the creation and analysis of finite-state models. SUPREMICA includes an automata simulator. It is possible to view the current state, choose which event to execute, observe the resulting state changes, and step forwards and backwards through the simulation. This makes it possible to compare the simulations of the learnt formal model and the simulation of the actual *LSM* code using MATLAB/Simulink.

Recall from Section 4.2 that the alphabet of the learnt DFA model is constructed using the input parameters of the *LSM* code. Therefore, executing the *LSM* code with a set of input parameters can analogously be simulated by executing the corresponding string of events in the DFA model. For instance, the result of executing the *LSM* code with the input parameter, `laneChangeRequest` set to `changeLeft` can be simulated in the DFA model by executing the event *changeLeft*. Since this comparison is made between the results from simulating the actual *LSM* implementation and the automata simulation of the learnt model, it would also validate the abstraction choices described in Section 4.1. Furthermore, a known existing bug in the *LSM* development code could be found to exist also in the learnt model. This was validated by manually simulating the learnt model with the sequence of in-

put parameter changes that produced the bug during the development testing phase of the actual *LSM* code.

Though we did not find any discrepancy between the code and the learnt model, such manual inspection is not exhaustive and therefore cannot guarantee completeness of the validation process. An alternative approach in this regard would be to use formal verification to verify the correctness. However, as only limited informal specifications (natural language) were available, this route was not (easily) pursued.

Still, the minimised model, together with the simulations in SUPREMICA strengthens the confidence in the results from the learning process. These models definitely serve as a formal documentation that provides insights into the decisions made during the design and implementation of the *LSM*.

Threats to Validity

In this case study, we investigated only one problem instance and so cannot make any concrete conclusions on the generalisation or the scalability of the approach. We could have accidentally chosen a piece of MATLAB code that lent itself particularly well to automatic learning. Indeed, we took a piece of code that we were already familiar with. Furthermore, the validation of the learnt model was admittedly rather superficial, visual inspection and comparison of simulation results between the learnt model and the actual MATLAB code. Ideally, we should have used the learnt model to assert functional properties of the MATLAB code. The closest we got in this respect was the known bug in the code, that could be shown to also be present in the learnt model.

However, we did use a general automata learning framework that was not tailored specifically to our case study; the only thing that was specifically implemented was the interface between the learning framework and MATLAB. Even so, that interface was intentionally kept general so that similar case studies of other pieces of code can be performed in the future to truly assess the validity of the presented approach.

5 Insights and Discussion

We have successfully learned a formal model of the *LSM* and have validated the model using multiple methods. In this section we present the insights gained from this case study and discuss directions for possible future research.

5.1 Towards Formal Software Development

The primary motivation for this work is to overcome the limitations in manual model construction so that techniques like formal verification and formal synthesis can be used to guarantee the correctness of software, which is especially important for autonomous driving, without disrupting current industrial practices. Insights from this case study can be used to scale active learning to obtain formal models for safety-critical software development. The presented approach is independent of the semantics of the implementation languages and therefore can be a valuable tool to obtain formal models from automotive software. The only requirement to seamlessly integrate this approach with the daily engineering workflow is the possibility to establish an interface between the production code and the learning algorithms. Such a seamless integration makes it easier to use formal methods not only for safety-critical software but also for other automotive software (e.g. infotainment).

In this paper, we only focus on automatically learning a formal model of the *LSM*, and do not focus on any kind of formal analysis on the learnt model. Formal analysis using different verification and synthesis tools can directly be done on the models learnt using SUPREMICA, in a similar way as presented in [16], [52]. Of course, the learnt models must be translated into an input format suitable for the particular tool.

Continuous Formal Development

With increasing complexity, software development in the automotive industry is adopting new model-based development approaches in the software development life cycle (SDLC) [53]–[56]. Quality assurance in such approaches relies on continuous integration methods where continuous testing is vital. However, safety critical software requires strict measures that are guaranteed only by formal approaches. Continuous formal verification [57] is a viable solution in this regard. Though there is a need for significant amount of research to adopt a continuous formal verification process for automotive SDLC, we think that active automata learning will play a major role in establishing such a workflow.

5.2 Practical Challenges

Interaction with the SUL

The interaction with real-life systems and the construction of application-specific learning setups remain as challenges for the automata learning community despite their application in different scenarios over the years [25], [39], [40]. In this case study, we successfully demonstrated the use of active automata learning in the context of embedded automotive software implemented in MATLAB, a new application area. To the best of our knowledge, this has not been done previously. However, in the course of our study, there were a few challenges.

A major aspect of the active learning process is to establish a proper interface between the learner and the SUL. In our case, the interface between the learner (Scala) and the SUL (MATLAB) is achieved through MATLAB-Java integration using the MATLAB Engine API for Java [50] as described in Section 4.2. As such an interface involves different type systems, the *dynamically* typed MATLAB and the *statically* typed Java and Scala, the risk of runtime type mismatch errors is increased. Another challenge is to establish an appropriate abstraction such that the learner can obtain necessary information about the alphabet to actively interact with the SUL. In this study, the aim is to learn a model of the *LSM*. Therefore, all external dependencies were abstracted such that the learner can easily interact with the SUL. However, we foresee that data dependencies between different methods and user defined classes could present additional challenges to scale this approach, for example to learn a model of the *Planner* and the *LSM* together.

In this case study, both these aspects, the interface and providing the required information to the learner (events, event predicates, etc.), were achieved manually. This effort needed to design and implement application specific learning setups can be reduced by creating test-drivers [39] in the form of standalone libraries and/or automatically constructing abstractions [23] for seamless integration between the SUL and the learner. Any work in that direction will make it easier to use popular learning frameworks like LearnLib [28], thereby strengthening the application of active automata learning to widespread industrial use within the automotive domain.

Efficiency of the learning algorithms

In this case study, we used two learning algorithms, the L^* and the MPL, to learn formal models from embedded automotive software. The implemented L^* algorithm failed to learn a complete model of the *LSM* due to two main issues: the exponential growth of the observation table and the equivalence queries. The problem with equivalence queries in this case study could be attributed to the use of W-method for counterexample search. While these issues are well known within the active learning community [25], we acknowledge that the use of other efficient implementations of the classic L^* algorithm that includes optimizations to address these issues could produce different results in this context. Efforts to benchmark such algorithmic implementations could be valuable in the long run.

The MPL is specifically developed to learn a modular system consisting of several interacting automata. The main benefit of such a modular algorithm is the reduction in search space of the learning process that is achieved by exploiting the structure of the SUL. Unfortunately, in this case study due to the structure of the *LSM*, we had to learn a monolithic model. However, we foresee that the modular approach could potentially be helpful in tackling the complexity that arises in learning larger systems. Research on defining suitable system architectures and appropriate learning abstractions remains to be done in this regard.

States vs events

Both the L^* and the MPL require a definition of the events that are relevant to the SUL. Interestingly, there is a trade-off between the size of the alphabet and the size of the state-space; small alphabet leads to large state-space, and vice versa. This trade-off is thus important, as the well known state-space explosion is a real practical problem.

The current learning setup resulted in 3072 events, one for each unique valuation of the input parameter, which was in the end reduced to 1536 events with satisfiable predicates. However, it is possible to use each of the input parameters as an event. This would result in a considerably smaller alphabet of only 23 events. Using the 23 events to learn leads to a huge state-space for both algorithms. For L^* we noticed that the number of equivalence queries significantly increases, and the MPL does not terminate as the state-space

becomes too large to handle.

Multiple interlaced lattice structures are seen in the partial models that were obtained and these relate to the various combinations of input parameters. The efficiency of the learning algorithms can be tweaked by leveraging this trade-off when abstracting the code.

5.3 Software Reengineering and Reverse Engineering

Reverse engineering, which involves extracting high level specifications from the original system can help to understand (ill-documented) legacy systems and black-box systems and to reason about the correctness of the complete software in complex systems. In addition, the development of intelligent autonomous driving features typically undergoes several design iterations before public deployment. In such a case, the formal approaches used to guarantee correctness need to conform to the software reengineering process. Reengineering embedded automotive software is different from software reengineering in other domains due to unique challenges [58], [59]. Experience from 10 years of software reengineering activities at one of the big automotive suppliers [60] reports that reengineering even a single module is expensive and it emphasizes the need for methods and tools tailored to the automotive domain.

The formal models learnt through active learning can help identify unintended changes between different software implementations during the software reengineering phase. Also, during the reverse engineering phase active learning can obtain high-level models from legacy systems and thereby help to understand and reason about them. The work in this paper can be used as a reference to identify possible future research in this regard.

6 Conclusion

In this paper, we have described a new, as far as we know, application area of active automata learning, namely interfacing with and learning MATLAB code. MATLAB/Simulink is currently the main engineering tool in the automotive industry, so by showing the applicability of active learning of MATLAB models, we take a huge step towards using formal approaches as a daily engineering tool. This is especially important for the development of safety-critical systems, like autonomous vehicles.

To the development code of a lane change module, *LSM*, being developed for autonomous vehicles, we applied two different active automata learning algorithms. One, an adaption of the well-known L^* algorithm. The other, an algorithm, MPL, designed for learning a modular model, though here it was used to learn a monolithic model due to the architecture of the *LSM*. L^* was unable to learn a model in 7 hours, and had memory issues, most likely due to the large alphabet and the use of the W-method to find counterexamples. MPL, on the other hand, having more information about the target system, learned a model in roughly one minute. Note though that our aim was not to benchmark L^* vs MPL, but to show a proof-of-concept for the use of active automata learning from MATLAB/Simulink code.

We validated the learnt model in four ways, which together made us confident that the learnt model was "correct":

- The language minimisation of the MPL model is very similar to the original meta-model of the *LSM*.
- Manual comparison of the learnt model to a manually developed model of the *LSM* indicated close similarity.
- Simulating the learnt automaton in Supremica and comparing to the simulation of the actual code in MATLAB/Simulink showed no obvious discrepancies.
- A known existing bug in the development code could be found to exist also in the learnt model.

Taken together, these make a strong argument for the usefulness of active automata learning in an industrial setting within the automotive domain.

Learning a monolithic model is a bottle neck as it scales badly. Learning modular models potentially allows to learn models of larger systems, which is important for industrial acceptance so this is clearly future research. Currently, the main obstacle is how to define the modules and partition the variables among the modules; if not done properly, the benefits of modular learning are lost.

There are existing learning frameworks, like LearnLib [28] and Tomte [23], that could potentially include more efficient algorithms than our own adaption of L^*. Furthermore, learning richer structures (but with the same expressive power), like extended finite state-machines [24], is definitely an interesting topic for further research.

All in all, our goal is to make active automata learning a tool to aid widespread adoption of formal methods in day to day software development within the automotive industry, in much the same way as MATLAB currently is.

References

[1] T. Litman, *Autonomous vehicle implementation predictions*. Victoria, Canada: Victoria Transport Policy Institute, 2020.

[2] Koopman, Philip and Wagner, Michael, "Autonomous vehicle safety: An interdisciplinary challenge", *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, 2017.

[3] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software", *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007.

[4] M. Broy, "Challenges in automotive software engineering", in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: Association for Computing Machinery, 2006, pp. 33–42, ISBN: 1595933751.

[5] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice", *Software & Systems Modeling*, vol. 17, no. 1, pp. 91–113, 2018.

[6] A. Charfi Smaoui, F. Liu, and C. Mraidha, "A Model Based System Engineering Methodology for an Autonomous Driving System Design", in *25th ITS World Congress*, Copenhagen, Denmark: HAL, Sep. 2018.

[7] P. Struss and C. Price, "Model-based systems in the automotive industry", *AI magazine*, vol. 24, no. 4, pp. 17–17, 2003.

[8] J. Friedman, "Matlab/simulink for automotive systems design", in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 1, Munich, Germany: IEEE, 2006, pp. 1–2.

[9] H. Altinger, F. Wotawa, and M. Schurius, "Testing methods used in the automotive industry: Results from a survey", in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, ser. JAMAICA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 1–6, ISBN: 9781450329330.

[10] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?", *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[11] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey", *ACM Computing Surveys (CSUR)*, vol. 52, no. 5, pp. 1–41, 2019.

[12] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey", *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, 2017.

[13] The MathWorks Inc., *Matlab products*, 2020.

[14] F. Leitner-Fischer and S. Leue, *Simulink design verifier vs. spin: A comparative case study*, 2008.

[15] M. Schurenberg, *Scalability analysis of the simulink design verifier on an avionic system*, 2012.

[16] Y. Selvaraj, W. Ahrendt, and M. Fabian, "Verification of decision making software in an autonomous vehicle: An industrial case study", in *Formal Methods for Industrial Critical Systems*, Cham: Springer International Publishing, 2019, pp. 143–159.

[17] A. Mashkoor, F. Kossak, and A. Egyed, "Evaluating the suitability of state-based formal methods for industrial deployment", *Software: Practice and Experience*, vol. 48, no. 12, pp. 2350–2379, 2018.

[18] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

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

[20] X. Liu, H. Yang, and H. Zedan, "Formal methods for the re-engineering of computing systems: A comparison", in *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, Washington, D.C.: IEEE, 1997, pp. 409–414.

[21] D. Angluin, "Learning regular sets from queries and counterexamples", *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[22] B. Steffen, F. Howar, and M. Merten, "Introduction to active automata learning from a practical perspective", in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, Berlin, Heidelberg: Springer, 2011, pp. 256–296.

[23] F. Aarts, "Tomte: Bridging the gap between active learning and real-world systems", PhD thesis, [Sl: sn], 2014.

[24] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines", *Formal Aspects of Computing*, vol. 28, no. 2, pp. 233–263, 2016.

[25] F. Howar and B. Steffen, "Active automata learning in practice", in *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, Cham: Springer International Publishing, 2018, pp. 123–148.

[26] A. Farooqui, F. Hagebring, and M. Fabian, *Active learning of modular plant models*, To appear, 2020.

[27] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010.

[28] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib", in *International Conference on Computer Aided Verification*, Cham: Springer International Publishing, 2015, pp. 487–495.

[29] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, *et al.*, "Bandera: Extracting finite-state models from java source code", in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, Limerick, Ireland: IEEE, 2000, pp. 439–448.

[30] G. J. Holzmann, "From code to models", in *Proceedings Second International Conference on Application of Concurrency to System Design*, Newcastle upon Tyne, UK: IEEE, 2001, pp. 3–10.

[31] G. J. Holzmann and M. H. Smith, "A practical method for verifying event-driven software", in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, Los Angeles, CA, USA: IEEE, 1999, pp. 597–607.

[32] R. Reicherdt and S. Glesner, "Formal verification of discrete-time matlab/simulink models using boogie", in *International Conference on Software Engineering and Formal Methods*, Cham: Springer International Publishing, 2014, pp. 190–204.

[33] D. Araiza-Illan, K. Eder, and A. Richards, "Formal verification of control systems' properties with theorem proving", in *2014 UKACC International Conference on Control (CONTROL)*, Loughborough, UK: IEEE, 2014, pp. 244–249.

[34] H. Fang, J. Guo, H. Zhu, and J. Shi, "Formal verification and simulation: Co-verification for subway control systems", in *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*, Beijing, China: IEEE, 2012, pp. 145–152.

[35] B. Jonsson, "Learning of automata models extended with data", in *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 327–349, ISBN: 978-3-642-21455-4.

[36] F. Aarts, J. Schmaltz, and F. Vaandrager, "Inference and abstraction of the biometric passport", in *Leveraging Applications of Formal Methods, Verification, and Validation*, T. Margaria and B. Steffen, Eds., Berlin, Heidelberg: Springer, 2010, pp. 673–686, ISBN: 978-3-642-16558-0.

[37] F. Aarts, J. De Ruiter, and E. Poll, "Formal models of bank cards for free", in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, Luxembourg: IEEE, 2013, pp. 461–468.

[38] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software", in *Formal Methods and Software Engineering*, M. Butler, S. Conchon, and F. Zaïdi, Eds., Cham: Springer International Publishing, 2015, pp. 67–83, ISBN: 978-3-319-25423-4.

[39] M. Merten, M. Isberner, F. Howar, B. Steffen, and T. Margaria, "Automated learning setups in automata learning", in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 591–607.

[40] M. Merten, "Active automata learning for real life applications", PhD thesis, TU Dortmund University, 2013.

[41] S. Kunze, W. Mostowski, M. R. Mousavi, and M. Varshosaz, "Generation of failure models through automata learning", in *2016 Workshop on Automotive Systems/Software Architectures (WASA)*, Venice, Italy: IEEE, 2016, pp. 22–25.

[42] M. Shahbaz, K. C. Shashidhar, and R. Eschbach, "Iterative refinement of specification for component based embedded systems", in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 276–286, ISBN: 9781450305624.

[43] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. New York, NY: Springer Science & Business Media, 2009.

[44] H. Zhang, L. Feng, and Z. Li, "A learning-based synthesis approach to the supremal nonblocking supervisor of discrete-event systems", *IEEE Trans. on Automatic Control*, vol. 63, no. 10, pp. 3345–3360, Oct. 2018, ISSN: 0018-9286.

[45] A. Farooqui and M. Fabian, "Synthesis of supervisors for unknown plant models using active learning", in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, Vancouver, BC, Canada: IEEE, 2019, pp. 502–508.

[46] T. Chow, "Testing software design modeled by finite-state machines", *IEEE Trans. on Software Engineering*, vol. 4, no. 03, pp. 178–187, 1978, ISSN: 0098-5589.

[47] F. B. Khendek, S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models", *IEEE Transactions on software engineering*, vol. 17, no. 591-603, pp. 10–1109, 1991.

[48] The MathWorks Inc., *Matlab*, 2020.

[49] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. California: Artima Inc, 2008.

[50] The MathWorks Inc., *Java engine api summary*, 2020.

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

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

[53] A. Rausch, O. Brox, A. Grewe, M. Ibe, S. Jauns-Seyfried, C. Knieke, M. Körner, S. Küpper, M. Mauritz, H. Peters, *et al.*, "Managed and continuous evolution of dependable automotive software systems", in *Proceedings of the 10th Symposium on Automotive Powertrain Control Systems*, Braunschweig: Cramer, 2014, pp. 15–51.

[54] M. Patil and S. Annamaneni, "Model based system engineering (mbse) for accelerating software development cycle", L&T Technology Services White Paper, Tech. Rep., 2015.

[55] K. Kubiček, M. Čech, and J. Škach, "Continuous enhancement in model-based software development and recent trends", in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Zaragoza, Spain: IEEE, 2019, pp. 71–78.

[56] K. M. Cie, "Agile in automotive–state of practice 2015", *Study, Kornwestheim*, p. 58, 2015.

[57] F. R. Monteiro, M. Y. R. Gadelha, and L. C. Cordeiro, "Boost the impact of continuous formal verification in industry", *CoRR*, vol. abs/1904.06152, 2019.

[58] A. Thums and J. Quante, "Reengineering embedded automotive software", in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, Italy: IEEE, 2012, pp. 493–502.

[59] V. Schulte-Coerne, A. Thums, and J. Quante, "Challenges in reengineering automotive software", in *2009 13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, Germany: IEEE, 2009, pp. 315–316.

[60] J. Quante, "Reengineering automotive software at bosch", *Softwaretechnik-Trends*, vol. 31, no. 2, 2011.