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

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

Yuvaraj Selvaraj[1,2(✉)], Wolfgang Ahrendt[2], and Martin Fabian[2]

[1] Zenuity AB, Gothenburg, Sweden
yuvaraj.selvaraj@zenuity.com
[2] Chalmers University of Technology, Gothenburg, Sweden
{ahrendt,fabian}@chalmers.se

**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.

**Keywords:** Autonomous driving · Formal verification · Supervisory Control Theory · Model checking · Deductive verification

## 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.

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

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-27008-7_9

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 imperative 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 [14,23].

The international standard ISO 26262 [16] 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 [13] 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 interdisciplinary 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 [30,34], Model Checking with Temporal Logic of Actions [22], Deductive Verification with contract based programming [4]. 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 [23] 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 [33], an overview of the challenges in designing, specifying and verifying cyber-physical systems, particularly semi-autonomous driving systems with human interaction is provided. [12] 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 [14,23], surveys on tool based verification methods and tools [5,9], and the general industrial adoption of formal methods technology [17,18,32,35].

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 Sect. 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 Sect. 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 [36], 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 [36].

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.

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

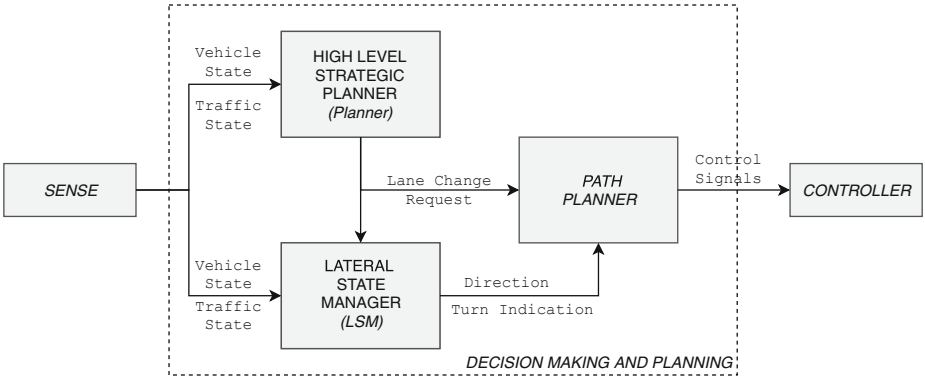


**Fig. 1.** System overview and interactions.

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 [36], 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 [5,9], the choice of the tools discussed in this paper is primarily motivated by prior case studies with Supremica [25,36], TLA$^+$ [20,27], and SPARK [1,7] on software systems similar in nature and scale to autonomous driving systems.

## 3 Supervisory Control Theory

The Supervisory Control Theory [31] (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 [34].

**Definition 1.** *An Extended Finite State Machine (EFSM) is a tuple* $E = \langle \Sigma, V, L, \rightarrow, L^i, L^m \rangle$*, where* $\Sigma$ *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 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

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

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 $E_1 \parallel E_2$ [34]. As defined by [34], 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* [26], which has shown remarkable efficiency and manages to handle systems of industrially interesting sizes and complexity. It can be shown [26] that when $\mathcal{E}$ is blocking, this is due to some *conflict* between the components of $\mathcal{E}$. Thus, the approach of [26] 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 [24].

### 3.2 Verification of *LSM* in Supremica

The software tool Supremica [25] implements the nonblocking verification algorithms mentioned above (as well as various other algorithms, both for verification and synthesis). To verify whether *LSM* presented in Sect. 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 [36], is illustrated with a small excerpt from the actual MATLAB-code, shown in Listing 1.1 with some variable and state names anonymized. Listing 1.1 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 Sect. 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*

**Listing 1.1.** 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
```

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.

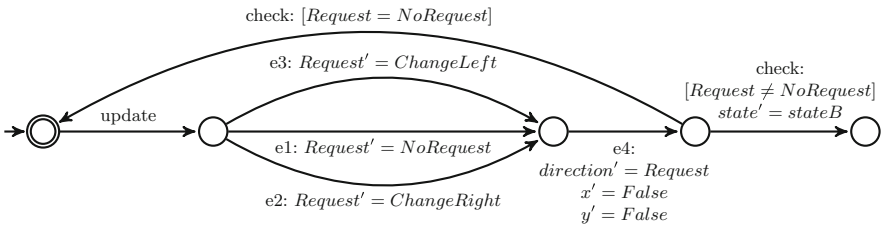


**Fig. 2.** EFSM of Listing 1.1. Primed variables represent next-state values.

*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` 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.

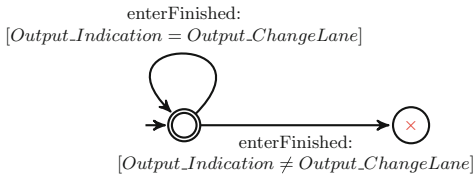


**Fig. 3.** EFSM of the specification to model *Req.1*. The blocking state is represented with a cross inside.

## 4 Model Checking

Model checking [10,29] is a framework for verification of finite transition systems using temporal logic [28] as a specification formalism. Several formalisms and powerful model checking tools have emerged over the years [6,11].

**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 [21]. 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 [20–22].

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

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

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 $\sigma$, iff *Init* is true in the first state of $\sigma$ and every step in $\sigma$ 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 (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 [20].

### 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 1.2 shows the TLA$^+$ translation of the MATLAB-code in Listing 1.1 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 1.1. A call to the function `duringStateA` is translated to a behaviour where the formula `During_StateA` is valid.

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 1.2. With the complete TLA$^+$ translation of the *LSM*, TLC

**Listing 1.2.** TLA$^+$ translation of the code in Listing 1.1.

```
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
```

can model check for desired properties, which are described using pre-defined statements and constructs available. More details on the statements and the restrictions on TLC is available in [22]. In order to verify *Req.1* of Sect. 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

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

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 [15] 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 [19], Java [3], or Ada [7].

### 5.1 SPARK

Ada [8] 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 [1]. SPARK is a subset of Ada with additional features to support formal verification [7]. 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 [1] 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 1.3 shows how the code in Listing 1.1 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.

**Listing 1.3.** SPARK implementation of the code in Listing 1.1.

```
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;
```

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 1.3 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 (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 [2,7].

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 (3) of a subprogram responsible for execution on the completion of a lane change.

```
Post => (Var.State = Finished) and
  (Output_Indication = Output_ChangeLane)          (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 (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 Sect. 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 mod-

ular 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 conventional 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. Adacore. https://www.adacore.com/. Accessed 26 Apr 2019
2. Spark 2014 reference manual. https://docs.adacore.com/spark2014-docs/html/lrm/index.html. Accessed 26 Apr 2019
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification-The KeY Book. LNCS, vol. 10001. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science. Springer, London (2009). https://doi.org/10.1007/978-1-84882-745-5
5. Armstrong, R.C., Punnoose, R.J., Wong, M.H., Mayo, J.R.: Survey of existing tools for formal verification. SANDIA REPORT SAND2014-20533 (2014)
6. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
7. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
8. Barnes, J.: Programming in Ada 2012. Cambridge University Press, Cambridge (2014)
9. Beckert, B., Hähnle, R.: Reasoning and verification: state of the art and current trends. IEEE Intell. Syst. **29**(1), 20–29 (2014)
10. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
11. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
12. Fisher, M., Dennis, L.A., Webster, M.P.: Verifying autonomous systems. Commun. ACM **56**(9), 84–93 (2013)
13. Forsberg, K., Mooz, H.: The relationship of system engineering to the project cycle. In: INCOSE International Symposium, vol. 1. Wiley Online Library (1991)
14. Guiochet, J., Machin, M., Waeselynck, H.: Safety-critical advanced robots: a survey. Robot. Auton. Syst. **94**, 43–52 (2017)
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580, 583 (1969)
16. ISO: Road vehicles - Functional safety. Technical report, ISO 26262 (2011)
17. Kasauli, R., Knauss, E., Kanagwa, B., Nilsson, A., Calikli, G.: Safety-critical systems and agile development: a mapping study. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE (2018)
18. Kemmerer, R.A.: Integrating formal methods into the development process. IEEE Softw. **7**(5), 37–50 (1990)
19. Kosmatov, N., Prevosto, V., Signoles, J.: A lesson on proof of programs with Frama-C. Invited tutorial paper. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 168–177. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_10
20. Lamport, L.: The TLA$^+$. https://lamport.azurewebsites.net/tla/tla.html. Accessed 22 Apr 2019
21. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. (TOPLAS) **16**(3), 872–923 (1994)

22. Lamport, L.: Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
23. Luckcuck, M., Farrell, M., Dennis, L., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: a survey. arXiv preprint arXiv:1807.00048 (2018)
24. Malik, R.: Programming a fast explicit conflict checker. In: 2016 13th International Workshop on Discrete Event Systems (WODES), pp. 438–443. IEEE (2016)
25. Malik, R., Akesson, K., Flordal, H., Fabian, M.: Supremica-an efficient tool for large-scale discrete event systems. IFAC-PapersOnLine **50**(1), 5794–5799 (2017). https://doi.org/10.1016/j.ifacol.2017.08.427. 20th IFAC World Congress
26. Mohajerani, S., Malik, R., Fabian, M.: A framework for compositional nonblocking verification of extended finite-state machines. Discrete Event Dyn. Syst. **26**(1), 33–84 (2016)
27. Newcombe, C.: Why Amazon chose TLA$^+$. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 25–39. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43652-3_3
28. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), pp. 46–57. IEEE (1977)
29. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
30. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
31. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989)
32. Saiedian, H., Hinchey, M.G.: Challenges in the successful transfer of formal methods technology into industrial applications. Inf. Softw. Technol. **38**(5), 313–322 (1996)
33. Seshia, S.A., Sadigh, D., Sastry, S.S.: Formal methods for semi-autonomous driving. In: 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE (2015)
34. Skoldstam, M., Akesson, K., Fabian, M.: Modeling of discrete event systems using finite automata with variables. In: 2007 46th IEEE Conference on Decision and Control, pp. 3387–3392. IEEE (2007)
35. Wolff, S.: 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, pp. 23–29. IEEE Press (2012)
36. Zita, A., Mohajerani, S., Fabian, M.: Application of formal verification to the lane change module of an autonomous vehicle. In: 2017 13th IEEE Conference on Automation Science and Engineering (CASE), pp. 932–937. IEEE (2017)