electronics

Article

Automatically Learning Formal Models from Autonomous Driving Software

Yuvaraj Selvaraj [1,2,*,†], Ashfaq Farooqui [2,†], Ghazaleh Panahandeh [1], Wolfgang Ahrendt [3] and Martin Fabian [2]

[1] Zenseact, 417 56 Gothenburg, Sweden; ghazaleh.panahandeh@zenseact.com
[2] Department of Electrical Engineering, Chalmers University of Technology, 412 96 Gothenburg, Sweden; ashfaqf@chalmers.se (A.F.); fabian@chalmers.se (M.F.)
[3] Department of Computer Science and Engineering, Chalmers University of Technology, 412 96 Gothenburg, Sweden; ahrendt@chalmers.se
* Correspondence: yuvaraj.selvaraj@zenseact.com
† These authors contributed equally to this work.

Abstract: The correctness of autonomous driving software is of utmost importance, as incorrect behavior may have catastrophic consequences. Formal model-based engineering techniques can help guarantee correctness and thereby allow the safe deployment of autonomous vehicles. However, challenges exist for widespread industrial adoption of formal methods. One of these challenges is the model construction problem. Manual construction of formal models is time-consuming, error-prone, and intractable for large systems. Automating model construction would be a big step towards widespread industrial adoption of formal methods for system development, re-engineering, and reverse engineering. This article applies *active learning* techniques to obtain formal models of an existing (under development) autonomous driving software module implemented in MATLAB. This demonstrates the feasibility of automated learning for automotive industrial use. Additionally, practical challenges in applying automata learning, and possible directions for integrating automata learning into the automotive software development workflow, are discussed.

Keywords: autonomous driving; active learning; formal methods; model-based engineering; automata learning

Citation: Selvaraj, Y.; Farooqui, A.; Panahandeh, G.; Ahrendt, W.; Fabian, M. Automatically Learning Formal Models from Autonomous Driving Software. *Electronics* **2022**, *11*, 643. https://doi.org/10.3390/electronics11040643

Academic Editors: Chuan Hu, Umar Zakir Abdul Hamid and Argyrios Zolotas

Received: 28 December 2021
Accepted: 15 February 2022
Published: 18 February 2022

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 behavior is paramount, as incorrect behavior 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 include non-technical aspects (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 [5–8].

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 manufacturers. The model-based approach to design, test, and integrate software systems is instrumental in achieving the necessary correctness guarantees for such complex systems. In this vein, several tools and methods have been developed and used over the years. In particular, the usage of MATLAB/Simulink [9] has become increasingly successful in a number of automotive companies [10].

A direct consequence of such 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 [11]) is currently a prominent technique for software quality assurance [12]. However, an approach only based on testing is insufficient and partly infeasible to guarantee the correctness of autonomous vehicles [13]. Thus, there is a need for strict measures for quality assurance, and the use of formal methods in this regard promise to be beneficial [14,15].

Formal verification techniques can indeed be used to identify design errors in MATLAB/Simulink function block networks using Simulink Design Verifier (SDV) [16], and Polyspace [17] to perform static code analysis on C code generated from the function block networks. However, there are limitations in SDV; for instance in scalability and in the verification of temporal properties that cannot be expressed as assertions [18,19]. SDV also exclusively works with Simulink, which presents a challenge in reasoning about MATLAB code without Simulink function blocks. Additionally, SDV and Polyspace cannot be used to reason about function blocks where the internal implementation details are unavailable. Thus, in such cases there is a need to use complementary methods to guarantee the correctness of the complete system under design.

In [20], 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 more beneficial if introduced during the early stages of the software development workflow rather than being used for after-development verification. However, there are several obstacles that impede the widespread adoption of formal methods [14,21]. Significant trade-offs (e.g., tools compatible with formal methods, development cost and time) have to be made that disrupt 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 [22]—to prove the absence of errors in software designs—or, formal synthesis techniques like supervisor synthesis [23]—to generate a controller/supervisor that is correct by construction—require a model that describes the behavior of the system. However, constructing a formal model that captures the behavior 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 specification and consequently the implementation changes frequently, especially so for rapidly evolving systems typical for the autonomous driving domain.

Automating model construction could speed up industrial adoption of formal methods by reducing the burden of manually constructing the models. Such automated methods will 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 use [21,24]. Automatically constructing formal models can also help understand and reason about ill-documented legacy systems and black-box systems, which is crucial for quality assurance of large-scale and complex automotive systems.

Active automata learning [25–32] is a field of research that addresses the problem of automatic model construction. These approaches constitute a class of machine learning algorithms that actively interact with the target system to deduce a finite-state automa-

ton [33] describing its behavior. This article is an extended version of [34], in which we reported results from a case study to automatically learn a formal model of the Lateral State Manager (LSM), a sub-component of an autonomous driving software (under development) programmed in MATLAB. In addition, we described an interface between the learning tool MIDES [35] and MATLAB, which was used to learn a model of the LSM using two active learning algorithms L^* and Modular Plant Learner (MPL). The results demonstrated the feasibility of our approach, but also the practical challenges encountered. This article extends our previous work [34] with:

- Extensive description of the learning algorithms L^* [36] and MPL [30] with illustrative examples.
- An evaluation on the practical applicability of L^* to automatically learn a model of the LSM is presented based on experiments using LearnLib, an open-source automata learning framework [32], in addition to MIDES.
- Analysis of the learning outcome is performed by investigating optimizations that can potentially help improve the practical applicability of the learning algorithms.
- New insights on the approach of using automata learning to enable formal automotive software development is presented.

Note that this article does not aim to compare the performance of the different algorithms, but to show the applicability of active automata learning in a MATLAB development environment. The experiments also showed that a known bug existing in the actual MATLAB code was present in the learned model as well. This is very much desired, because the analysis of a model can only reveal a bug in the system if the bug also manifests itself in the model.

This article is structured as follows. Section 1.1 presents a brief overview of related work. Section 2 describes the system under learning followed by the necessary preliminaries in Section 3. Section 4 illustrates the learning algorithms with an example. A description of the learning framework and the results from the learning are presented in Sections 5 and 6, respectively. Section 7 presents the evaluation of the results. Section 8 includes the validation of the formal model learned and the threats to validity. Section 9 presents some insights from the experiments, discusses practical challenges and possible directions for integrating automata learning into automotive software development workflow. The article is concluded in Section 10 with a summary and some ideas for future work.

1.1. Related Work

Automatically learning finite-state models for formal verification has been done previously, for instance, from Java source code in [37], and from C code in [38,39]. These methods rely on extracting an automaton by analyzing the program source code. Hence, they are specific to the particular programming language and strictly rely on well defined coding patterns and program annotations. Additionally, the approaches of [37–39] cannot extract models where the source code is not available, such as when dealing with black-box systems or binaries. Active automata learning mitigates these restrictions and learns models of black-box systems through interaction.

There exist works on integrating the MATLAB development environment with tools compatible for formal verification. For example, in [40], MATLAB/Simulink designs are translated to the intermediate language Boogie that can later enable the use of SMT solvers [41] for verification. Other works include developing MATLAB toolboxes to integrate with a theorem prover [42] and a hybrid model checker [43]. Such methods depend on considerable manual (and skilled) work to understand the semantics of the MATLAB commands and built-in functions to develop the respective toolboxes. In contrast, by actively interacting with the actual MATLAB code the work in this article learns a formal model, which allows us to use general purpose formal methods tools to asses properties of the code. In addition, knowledge about the semantics of MATLAB code are not needed by the learning tool.

Active automata learning has been successfully applied to learn and verify communication protocols using Mealy machines [26,44], and to obtain formal models of biometric passports [45] and bank cards [46]. In [47], 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 [29,48,49]. There are very limited examples on the use of active automata learning in an automotive context [50,51] and it is yet to find its place in automotive software development. To the best of the authors' knowledge, active automata learning has not been used previously to learn formal models from MATLAB.

2. System under Learning: The LSM

The system under learning (SUL), the LSM, is a sub-component of the decision making and planning module in an autonomous driving system and is responsible for managing modes during an autonomous lane change. The lane change module is implemented in MATLAB-code [9] using several classes with different responsibilities. A simplified overview of the system and the interaction of the LSM with a high level strategic *Planner* and a low level *Path Planner* is shown in Figure 1. The lane change module is cyclically updated with the current vehicle state (e.g., position, velocity), surrounding traffic state, and other reference signals.

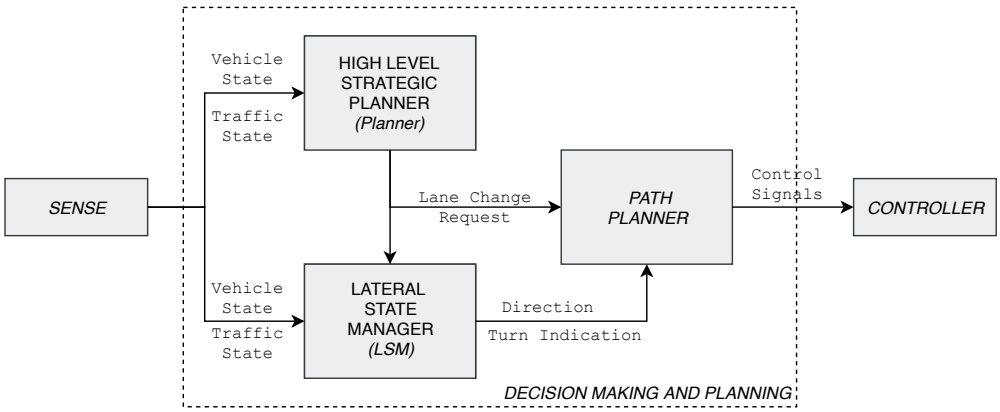


Figure 1. The lane change module system overview and interactions between the three components: *Planner*, *LSM*, and *Path Planner*.

The *Planner* in the lane change module is responsible for strategic decisions. Depending on the state of the vehicle, the *Planner* sends lane change requests to the *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, 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* plans a path and sends required control signals to the low level controller to perform a safe and efficient lane change. Once a lane change is initiated, the LSM needs to remember where in the sequence it is, thus it is implemented as a finite state machine. A representation 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*, abbreviated as S_F in Figure 2, which represents the completion of the lane change process.

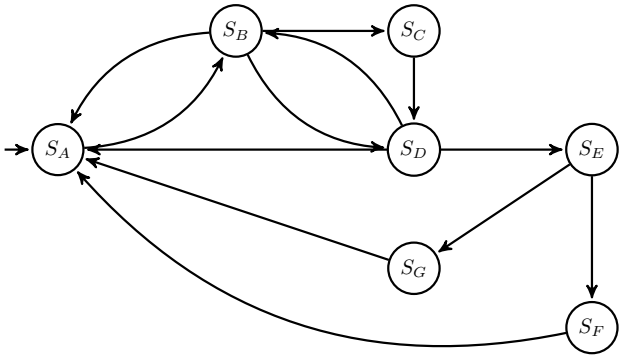


Figure 2. A finite state machine representing the LSM.

A call to the LSM is issued at every update cycle. During each call, the LSM undergoes three distinct execution stages. In the first stage, an associated function `updateState` is executed that updates all the inputs according to the function call arguments. Then, 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.

3. Preliminaries

An *alphabet*, denoted by Σ, is a finite, nonempty set of events. A *string* is a finite sequence of events chosen from Σ. The empty string, denoted by ε, is the string with zero events. For two strings, s and t, their *concatenation* denoted by st, is the string formed such that s is followed by t. Let Σ^k be the set of all strings over Σ of length k. Then, Σ^2 is $\Sigma\Sigma$ 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 $u = st$; t is then a *suffix* of u. 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 an ordered set of* state variables*, where each variable* v_i *has a discrete finite domain defined as* v_i^D*. A* state *is then defined as the assignment of values to variables,* $\hat{V} \in V^D$ *where* $V^D = v_1^D \times v_2^D \times \cdots \times v_n^D$*.* $\hat{V}$ *is called a* valuation*.*

Definition 2 (DFA)**.** *A* deterministic finite automaton *(DFA) 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 *of events;*
- $\delta : Q \times \Sigma \rightharpoonup Q$ *is the* partial transition function*;*
- $q_0 \in Q$ *is the* initial state*;*
- $M \subseteq Q$ *is the set of* marked states*.*

The set of all DFA is denoted $\mathcal{A}$. Every DFA $A \in \mathcal{A}$ determines a language *generated*, respectively, *marked*, by that DFA, defined with help of the extended transition function.

Definition 3 (Extended Transition Function)**.** *Given a DFA* $\langle Q, \Sigma, \delta, q_0, M \rangle$*, the* extended transition function $\bar{\delta} : Q \times \Sigma^* \rightharpoonup Q$ *is defined as (with* $s \in \Sigma^*$*,* $a \in \Sigma$*):*

- $\bar{\delta}(q, \epsilon) = q$
- $\bar{\delta}(q, sa) = q'$ *if there exists* $q'' \in Q$ *s.t.* $\bar{\delta}(q, s) = q''$ *and* $\delta(q'', a) = q'$

Definition 4 (Generated and Marked Language)**.** *Given a DFA* $A = \langle Q, \Sigma, \delta, q_0, M \rangle$*, the languages* generated *and* marked *by* A*,* $\mathcal{L}(A)$ *and* $\mathcal{L}_m(A)$*, respectively, are defined as:*

- $\mathcal{L}(A) = \{ s \in \Sigma^* \mid \bar{\delta}(q_0, s) \in Q \}$
- $\mathcal{L}_m(A) = \{ s \in \mathcal{L}(A) \mid \bar{\delta}(q_0, s) \in M \}$

Intuitively, the marked language is the set of all strings that lead to marked states. While the generated language denotes behavior that is possible but not necessarily accepted, the marked language denotes possible behavior that is accepted. A language is said to be *regular* if it is marked by some DFA. It is well-known [33] 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.

4. The Learning Algorithms

This section introduces and illustrates the learning algorithms used in this article. An example consisting of two robots, R1 and R2, is used as the SUL. Each robot can perform two operations, *load* and *unload*, represented by the events l_1 and u_1, respectively, for R1, and l_2 and u_2 for R2. The marked languages of R1 and R2 are $\mathcal{L}_m(R1) = (l_1 u_1)^*$ and $\mathcal{L}_m(R2) = (l_2 u_2)^*$, respectively. The behaviors of the robots represented as automata are shown in Figure 3, to the left. Each robot starts in its respective initial state `i`, and moves to the working state `w` on the occurrence of a *load* event. Then the robot transits back to its initial state on the occurrence of an *unload* event. An automaton representing their joint behavior is given in Figure 3c.

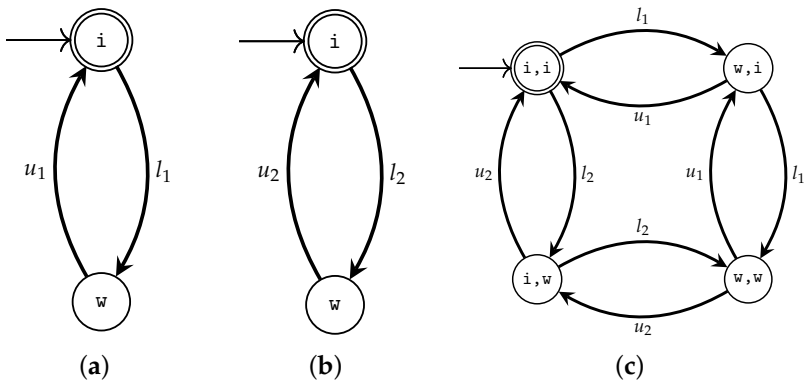


Figure 3. The example system. (**a**) R1; (**b**) R2; (**c**) joint behavior of R1 and R2.

4.1. The L^* Algorithm

The L^* algorithm [25] is a well-known active automata learning algorithm that has inspired a tremendous amount of work [26–29,32,36,46,52]. It learns a minimal automaton $\hat{\mathcal{M}}$ that generates the regular language $\mathcal{L}(\hat{\mathcal{M}}) \subseteq \Sigma^*$ representing the behavior of the SUL. L^* assumes access to an *oracle* that has complete knowledge of the system, and works by posing *queries* to the oracle. The modified L^* from [36,52] is used in this work. The learning algorithm interacts with the SUL to answer 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. Otherwise, 0 is returned. The membership query has the signature: $T : \mathcal{A} \times \Sigma^* \to \{ 0, 1, 2 \}$, and for $A \in \mathcal{A}$ and $s \in \Sigma^*$:

$$T(A, s) = \begin{cases} 2, & s \in \mathcal{L}_m(A) \\ 1, & s \in \mathcal{L}(A) \setminus \mathcal{L}_m(A) \\ 0, & s \notin \mathcal{L}(A) \end{cases} \tag{1}$$

Equivalence Queries: Given a *hypothesis* automaton $\mathcal{H}$, an algorithm verifies if $\mathcal{H}$ represents the language $\mathcal{L}(\hat{\mathcal{M}})$. If not, a *counterexample* $c \in \Sigma^*$ must be provided, such that, either c is incorrectly generated (that is, $c \in \mathcal{L}(\mathcal{H})$ but $c \notin \mathcal{L}(\hat{\mathcal{M}})$), or incorrectly rejected (that is, $c \notin \mathcal{L}(\mathcal{H})$ but $c \in \mathcal{L}(\hat{\mathcal{M}})$) by $\mathcal{H}$. In this work, equivalence queries are performed using the W-method [53].

Let $\hat{\mathcal{M}}$ have n states. Given a hypothesis $\mathcal{H}$, with $m \leq n$ states, the W-method creates test strings to iteratively extend the hypothesis until it has $n \geq m$ states.

The learner constantly updates its knowledge about the SUL's language as an *observation table*. The observation table $O(S, E, T)$ is a 2-dimensional table, where S is a set of prefix-closed strings, and E is the set of suffix-closed strings. The table has rows indexed by elements of $S \cup (S\,\Sigma)$, and the columns indexed by elements of E. The value of a cell (s, e) (for $s \in S \cup (S\,\Sigma)$ and $e \in E$) is populated using membership queries. The algorithm ensures that the observation table is closed and consistent [36]. The observation table is used to obtain a deterministic finite-state automaton, the hypothesis. Then, the learner performs an equivalence query on the hypothesis automaton. If a counterexample is found, it is added to the observation table together with all its prefixes. The algorithm loops until no counterexample can be found.

Example 1. *To illustrate the working of the L^* algorithm, consider the example with the two robots. L^* initializes the observation table as seen in Figure 4. The empty event ε is related to the initial state, which is marked, and hence its membership query results in a value of* 2*. On the other hand, the table entries for the l_1 and l_2 events are* 1*, as these events are defined from the initial state but do not reach marked states. Additionally, membership queries for strings that begin with u_1 and u_2 result in a value of* 0*, as they are not defined from the initial states. Hence, rows corresponding to such strings are not included in subsequent observation tables. The rows corresponding to the two sets of elements belonging to S and $S\,\Sigma$ are separated by a horizontal line in the table. For the sake of compactness, the ε prefix is omitted for non-empty strings.*

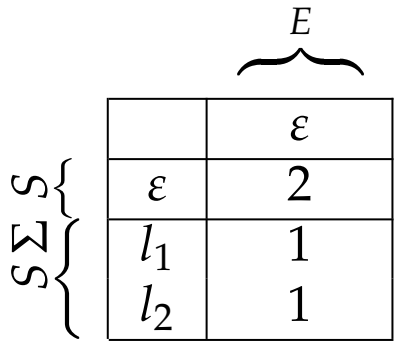


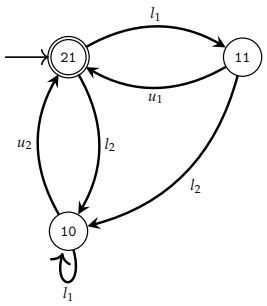
Figure 4. L^*—The initial table.

The initial table is made closed, and consistent using membership queries, and the resulting table is shown in Figure 5a and its corresponding automaton in Figure 5b. States in the automaton correspond to the row values of the table.

Given the first hypothesis, L^* now makes an equivalence query resulting in the counterexample $l_1 l_2 u_1$ According to the hypothesis, this string is rejected. However, the membership query for this string will result in a value 1*, as this string is possible in the system, as seen in Figure 3c. This counterexample and its prefixes are incorporated into the set S, and the learning continues until a new closed and consistent table, Figure 6a, is obtained. The corresponding hypothesis is seen in Figure 6b. Since no counterexample can be found at this stage, the algorithm terminates, returning the hypothesis as the learned model.*

	ε	l_2
ε	2	1
l_1	1	1
l_2	1	0
$l_2\ u_2$	2	1
$l_1\ u_1$	2	1
$l_1\ l_2$	1	0
$l_2\ l_1$	1	0

(a)

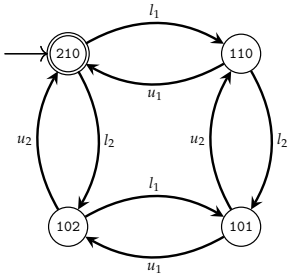


(b)

Figure 5. L^*—First iteration. (**a**) The initial table made closed and consistent; (**b**) first hypothesis.

	ε	l_2	u_2
ε	2	1	0
l_1	1	1	0
l_2	1	0	2
$l_2\ u_2$	2	1	0
$l_1\ l_2$	1	0	1
$l_1\ l_2\ u_1$	1	0	2
$l_1\ u_1$	2	1	0
$l_2\ l_1$	1	0	1
$l_2\ u_2\ l_1$	1	1	0
$l_2\ u_2\ l_2$	1	0	2
$l_1\ l_2\ u_2$	1	1	0
$l_1\ l_2\ u_1\ l_1$	1	0	1
$l_1\ l_2\ u_1\ u_2$	2	1	0

(a)



(b)

Figure 6. L^*—Second iteration. (**a**) The table updated with the counterexample and made closed and consistent; (**b**) second hypothesis.

4.2. The Modular Plant Learner

The Modular Plant Learner [30] is a state-based active learning algorithm developed to learn a *modular model*, that is, one composed of a set of interacting automata. These *modules* together define the behavior of the system. MPL does so by actively exploring the state-space of a program in a breadth-first search manner. It exploits structural knowledge of the SUL to search smartly. Hence, it requires access to the SUL's variables, and a Plant Structure Hypothesis (PSH) [30] defining the structure of the system. The PSH is a 3-tuple $H = \langle D, E, S \rangle$, where D is a set that provides a unique name for each module that is to be learned. The cardinality of D defines the number of modules to learn. E is an *event mapping* that defines which events belong to which module. S is a *state mapping* that defines the relationship between the modules and the variables in the SUL. The algorithm consists of the *Explorer*, which explores new states and a *ModuleBuilder* for each module to keep track of its 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 initialized with the SUL's initial state in the queue, which becomes the search's starting state. For each state in the queue, the *Explorer* checks if an event from the alphabet Σ can be executed. If a transition is possible, the *Explorer* sends the current state (q), the event (σ), and the state reached (q') to all the *ModuleBuilder*s.

Each of the *ModuleBuilder*s evaluates if the received transition is relevant to its particular module. If it is, the transition is added to the module; otherwise it is discarded. The

ModuleBuilder tracks the learning of each module as an automaton. This is done by maintaining a set Q_m containing the module's states and a transition function $T_m : Q_m \times \Sigma_m \rightharpoonup Q_m$, for each module $m \in D$. Once the transition is processed, the *ModuleBuilder* waits for the *Explorer* to send the next transition. The algorithm terminates when all modules are waiting and the exploration queue is empty. Each *ModuleBuilder* can now construct and return an automaton based on Q_m and T_m.

Example 2. *Consider again the example with the two robots. Assume that the robots' states are stored in the variables* $R1_{var}$ *and* $R2_{var}$*, respectively, represented as a state vector* $\langle R1_{var}, R2_{var} \rangle$*. The initial state is then* $\langle i, i \rangle$*. A PSH for this example is defined as follows:*

- $D = \{R1, R2\}$
- $E(R1) = \{l_1, u_1\}$
- $E(R2) = \{l_2, u_2\}$
- $S(R1) = \{R1_{var}\}$
- $S(R2) = \{R2_{var}\}$

At the start, the Explorer *knows only about the initial state. Two* ModuleBuilders *are initialized, one for each robot. The* ModuleBuilders *use the known initial state and knowledge regarding the PSH to initialize themselves as seen in Figure 7. The state marked blue in the* Explorer *denotes the state that is to be explored next.*

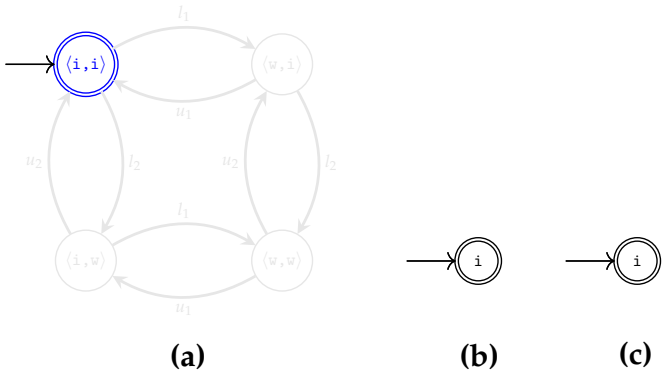


Figure 7. MPL—Initialization. (**a**) Explorer; (**b**) R1; (**c**) R2.

Once initialized, the Explorer *attempts to execute all the events in the alphabet from the initial state. Accepted states are reached only for the events* l_1 *and* l_2*; events* u_1 *and* u_2 *cannot be executed. Identified transitions are sent to the* ModuleBuilders*, where they are processed according to the PSH. Figure 8a shows the knowledge gained by the* Explorer*. The transitions in red show the current execution of the* Explorer*, and the blue states represent the states reached, but not yet explored. Figure 8b,c show the internal representation of the knowledge for each of the* ModuleBuilders*. The states reached in each of the modules are new. Hence, the corresponding global states are added to the exploration queue.*

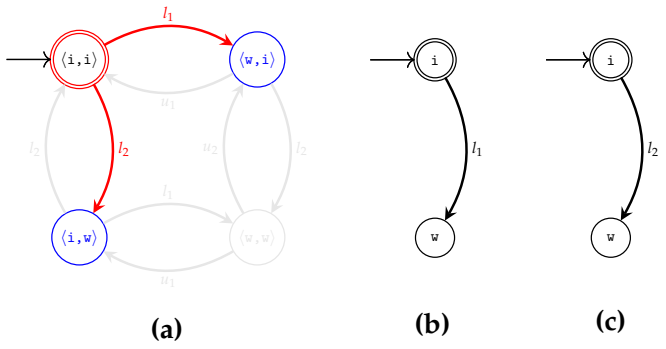


Figure 8. MPL—First Iteration. (**a**) Explorer; (**b**) R1; (**c**) R2.

For each of the states present in the queue, the Explorer *tries to execute all the events, and the obtained transitions, colored red in Figure 9a, are sent to the* ModuleBuilders*. The* ModuleBuilders *take only transitions relating to u_1 and u_2 to update their knowledge. As seen in Figure 9b,c the states reached by these (newly added) transitions are not new and have already been explored. Hence, no more states need to be explored, and the algorithm terminates. At termination the* Explorer *has explored only three states to learn a modular model describing the behavior of the two robots. In the worst-case scenario, though, the MPL must explore the entire state-space. This depends on the user-defined PSH and the possibility to decouple the system. Further details about the MPL are found in [30].*

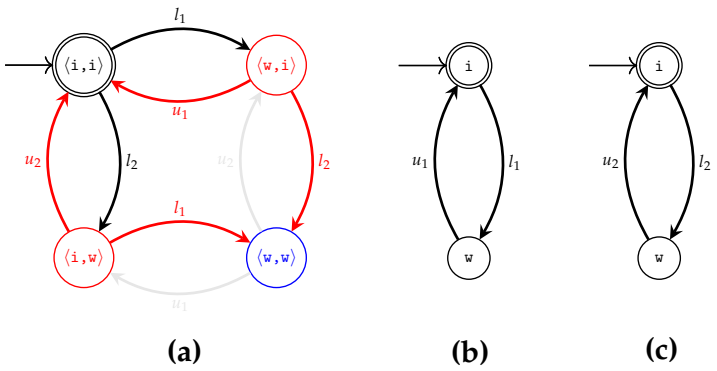


Figure 9. MPL—Final Iteration. (**a**) Explorer; (**b**) R1; (**c**) R2.

5. Method: The Learning Setup

To actively learn a DFA model of the SUL, an interface is necessary to execute (strings of) events, which represent the executable actions of the SUL. It should be possible to observe and set the state of the SUL. If an event is requested that is not executable by the SUL in its current state, the SUL should reply with an error message. Figure 10 presents an overview of the active automata learning setup used in this article. The learner refers to the learning tool MIDES [35] that implements the two learning algorithms described in Section 4. Furthermore, the learner can be replaced with other tools that follow a similar setup for automata learning, such as LearnLib [32]. The learning setup allows learning of automata models by (actively) interacting with the SUL. The following subsections describe the components and the learning outcome.

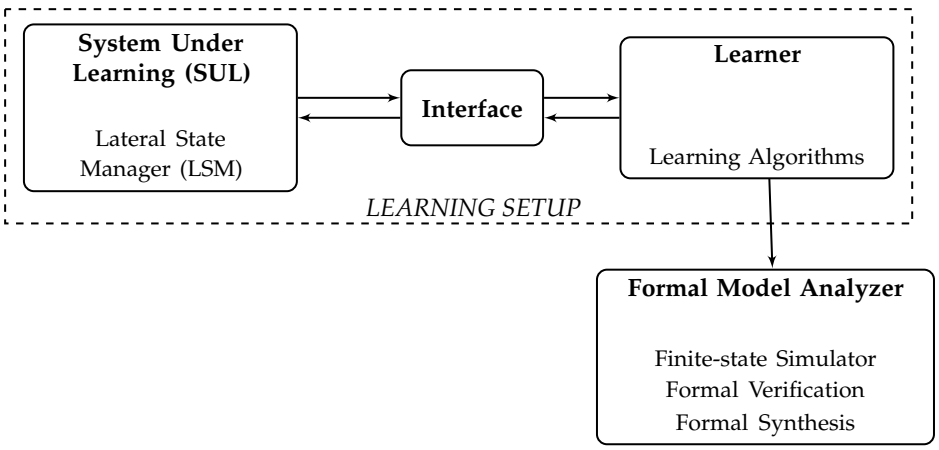


Figure 10. Overview of the learning setup.

5.1. Abstracting the Code

As described in Section 2, the LSM is a part of the lane change module, which is updated cyclically with the necessary signals. 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, which is described using Example 3.

Example 3. *Consider the small MATLAB code snippet shown in Listing 1. The function* `duringStateA` *decides whether the system transits to a new state or not depending on* `var1` *and* `var2`*. The values of these two variables are dependent on external function calls,* `function1` *and* `function2`*, respectively.*

Listing 1. A small illustrative example.

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

Listing 2 shows how the external function calls are replaced by the additional input argument `decisionVar`*. The two variables,* `decisionVar.var1` *and* `decisionVar.var2`*, have the domain* $\{True, False\}$*. While this abstraction is not universally valid and increases the number of input parameters of the function, it is possible in this particular context due to the way the different modules interact; the decision logic remains unchanged.*

Listing 2. Abstracted version of Listing 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 input parameter, `decisionVar`, 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 is used by the learner to observe the behavior of the LSM during their interaction, as described in the following section.

5.2. Interaction with the SUL

The interaction between the learning tool MIDES and the LSM, implemented in MATLAB, is crucial, for which there is a need to:

1. Enable communication between MIDES and MATLAB,
2. Provide information to MIDES on how to execute the LSM and observe the output.

MIDES must be able to call MATLAB functions, evaluate MATLAB statements, and pass data to and get data from MATLAB. In this learning setup, the learner is compiled to Java bytecode, and the resulting executable code is run on a Java virtual machine. Therefore, the interface integrates Java with MATLAB using the MATLAB Engine API for Java [54], providing a suitable API for MIDES to interact with MATLAB.

With this interface established, the learner can now call the `updateState` function by providing an input assignment to the corresponding variables. However, to learn a model,

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 via the `updateState` function, the input parameters define the alphabet of the model. Each unique valuation of the input parameters corresponds to one event in the alphabet. Since the abstracted LSM module is provided to the learner, each function that is abstracted into a decision variable potentially results in one additional input parameter. Following the abstraction described in Section 5.1, ten external function calls in the LSM resulted in ten Boolean valued `decisionVar`, in addition to one three-valued `laneChangeRequest`, as input parametersand a total of 3072 events. However, as state transitions in the LSM are defined only for a subset of these events, some of them would potentially not have any effect on the model behavior, 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 in Listing 2, 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 behavior of the LSM, the learner requires a set of variables given by the output of the `updateState` function. Furthermore, initial valuation of the variables, which is then the initial state of the LSM, is known to the learner.

6. Results

This section discusses the learning outcome. The learning algorithms were run on an Intel i7 machine, with 8 GB ram, running Linux.

6.1. Learning with L^*

The L^* algorithm implemented in MIDES ran out of memory during the experiments and could not learn a full model. In our longest learning experiment, after 13 h of learning, it was observed that 6 iterations of the hypothesis involving about 500k membership queries resulted in a hypothesis model with 8 states and 231 transitions. On visual inspection, the automaton structure resembles parts of Figure 2. Each of the states in the partially learned model correspond to one or more states in the automaton of Figure 2. However, since L^* did not terminate successfully, further analysis is needed.

Two main obstacles were faced while learning using the L^*. Firstly, as the observation table grows in size, it takes longer to make the table closed and consistent. Furthermore, the memory used to store the table grows rapidly by a factor dependent on the size of the alphabet. Secondly, an exhaustive search for a counterexample using the W-method in the given setup is time consuming. The number of test strings grows rapidly due to the large alphabet size, which slow down the equivalence queries. A detailed analysis on learning using the L^* algorithm is discussed in Section 7.

6.2. 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.

The resulting automaton consists of 37 states and 687 transitions. The learning took a total of 68 seconds. Furthermore, applying language minimization [33] to the learned model results in a model with 6 states and 114 transitions. The language minimized automation is

shown in Figure 11, and its similarity to Figure 2 is obvious. Multiple transitions between two states are indicated by a single transition in Figure 11. The two states S_G and S_F of Figure 2 have the same future behavior and hence are *bisimilar* [33], so they both correspond to the single state q_6 of Figure 11.

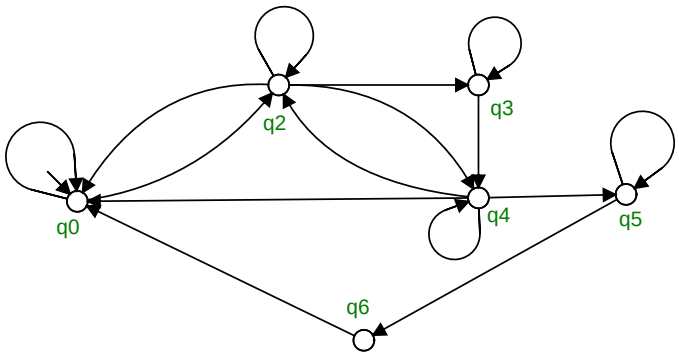


Figure 11. Learned model of the LSM.

The self-loops in the states of Figure 11 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 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 learned 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.

7. Evaluation

As described in Section 6.1, the L^* algorithm implemented in MIDES did not learn a complete model. Prior to performing a detailed analysis, it is essential to eliminate any potential implementation specific causes for this negative outcome. Typical causes in this regard could be related to the use of inefficient data structures or non-optimized search strategies. LearnLib is an open-source library for active automata learning that has been shown to outperform other existing open-source libraries [32]. LearnLib features a variety of automata learning algorithms including the L^* algorithm. As LearnLib is implemented in Java, it is possible to use an interface similar to the one in Section 5.2 to actively learn a model of the LSM.

The L^* algorithm implemented in LearnLib was run on a standard computer with 2.7 GHz Intel i5 processor and 8 GB of ram storage to learn a model of the LSM. However, even in this case, the L^* algorithm from LearnLib failed to learn a complete model of the LSM before it ran out of memory. With these two experimental results, there is adequate evidence to rule out any implementation specific causes for the negative outcome and to warrant further analysis as described in the next section.

7.1. Learning Complexity

The complexity of L^* depends on the number of membership and equivalence queries as they require interaction with the system and storage of information in the observation table. In [25], the theoretical worst case size of the observation table is calculated to be $(k+1)(n+m(n-1))n$, where $k = |\Sigma|$ is the size of the alphabet, $n = |Q|$ is the number of states, and m is the maximum length of any counterexample presented by the oracle. This is also the upper bound for the number of membership queries and thus the complexity is $\mathcal{O}(m|\Sigma||Q|^2)$. Assuming that $m = |Q|$ in the worst case, the number of membership queries (and the size of the observation table) is $\mathcal{O}(|\Sigma||Q|^3)$. Note from Section 6.2 that the minimal automaton had 6 states. Since L^* learns a minimal model, it is safe to assume that it would learn a model with 6 states. Taking these values as an example would mean

that learning the LSM would involve $1536 \times 6^3 \approx 3.3 \times 10^5$ membership queries. This corresponds with the observations in Section 6.1.

The efficiency of L^* for learning finite automata in practice has been investigated by empirical studies on some real-word examples and some randomly generated examples. For instance, [55] observes that the required number of membership queries grows quadratically in the number of transitions when learning prefix-closed languages. This is a challenge, as learning from autonomous driving (or automotive) software in general would typically involve models of reactive systems and therefore learning prefix-closed languages like the case with the LSM. Further empirical studies in [56] show that in general, more queries are required (i.e., harder) to learn a DFA with more marked states ($\approx|Q|$) and the number of membership queries per equivalence query grows linearly as a function of $|\Sigma|$ and $|Q|$. These results show the need for optimizations of the L^* algorithm for practical applicability, as is further shown by this study.

7.2. Alphabet Reduction

Based on the discussions so far, it is evident that optimizations are necessary to successfully learn a model of the LSM. The size of the alphabet directly affects the number of membership and equivalence queries. As also pointed out by [55], finding a counterexample to refute a hypothesis in an equivalence query becomes increasingly hard with large alphabets and prefix-closed languages. Therefore, any optimization that reduces the number of such queries is potentially beneficial. In this regard, it is valuable to investigate whether knowledge about the LSM could be exploited to reduce the size of the alphabet without loss of information during the learning process.

Through the abstraction method in Section 5, the alphabet size can be decreased by a reduction in the set of decision variables during the abstraction. For example, in Listing 2 the two variables `decisionVar.var1` and `decisionVar.var2` can potentially be combined into a single variable `decisionVar.var12`. Such a simplification is only possible if it is the case that `decisionVar.var1` and `decisionVar.var2` are exclusively used in `StateA` and do not affect other state changes in the LSM. This information could be obtained from knowledge about the design requirements of the LSM used to create the abstraction shown in Figure 2.

The above approach was implemented through a wrapper function before a call to the initially abstracted LSM from Section 5.2 is made. This ensures that the decision logic in the LSM remains unchanged. Thus, the number of Boolean decision variables was reduced from 10 to 6; bringing down the number of possible events to 192. However, even in this case, L^* failed to learn a complete model before running out of memory.

As further optimization was necessary, more knowledge about the LSM was used to reduce the alphabet size. The decision logic of the LSM was studied to find a correlation between the (now abstracted) 6 Boolean decision variables, showing that only 4 combinations were actually used, which could be encoded using only 2 Boolean variables. This resulted in 12 events in total, and L^* successfully learned a model similar to the one shown in Figure 11. The wrapper function described here to reduce the alphabet is similar to the approach of using event predicates described in Section 5.2 to remove unnecessary events from the alphabet. However, to reduce the alphabet in the proposed manner requires considerable knowledge about the LSM and is potentially prone to errors due to the manual abstraction involved. Though it is a limitation that the complete set of events is not queried in this way, it was adopted to explore the limits of L^*.

8. Model Validation

A formal model of the LSM was learned using the MPL and the optimized L^* as described in Section 6 and Section 7.2, respectively. To validate the learned model, similar to [45], it was compared to a model [57] manually constructed from the MATLAB code. This was done using the tool SUPREMICA [58], which 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. Thus, a comparison of the simulations of the learned formal model and the simulation of the actual LSM code using MATLAB/Simulink is made.

Recall from Section 5.2 that the alphabet of the learned 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. Since this comparison is made between the results from simulating the actual LSM implementation and the automata simulation of the learned model, it also validates the abstraction choices described in Section 5.1. Furthermore, a known existing bug in the LSM development code manifested itself also in the learned model. This was validated by manually simulating the learned model with a sequence of input parameter changes known to provoke the bug.

Though no discrepancy was found between the code and the learned model, such manual inspection is not exhaustive and cannot guarantee completeness of the validation process. Alternatively, formal verification could be used to verify correctness. However, as only limited informal (natural language) specifications were available, this was not (easily) done. Still, the minimized model, together with the simulations in SUPREMICA strengthens the confidence in the results of the learning process.

Threats to Validity

This article investigated only one problem instance and so cannot give any concrete conclusions on the generalization or the scalability of the approach. Accidentally, a piece of MATLAB code could have been chosen that lent itself particularly well to automatic learning. Indeed, a piece of code was chosen that the authors were already familiar with. Furthermore, the validation of the learned model was admittedly rather superficial, visual inspection and comparison of simulation results between the learned model and the actual MATLAB code. Ideally, the learned model should have been used to assert functional properties of the MATLAB code. The closest in this respect was the known bug in the code, that could be shown to also be present in the learned model.

However, a general automata learning framework that was not tailored specifically to the SUL in this article was used; 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.

9. Insights and Discussion

A formal model of the LSM was successfully learned, and validated in multiple ways. This section presents a discussion on the insights gained.

9.1. 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, without disrupting current industrial practices. The presented approach is independent of the semantics of the implementation languages One technical 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). Though this article does not consider any kind of formal analysis on the learned model, formal analysis using different verification and synthesis tools can directly be done on the learned models with tools like SUPREMICA, similar to [20,57]. Of course, the learned 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) [59–62]. Quality assurance in such approaches relies on continuous integration methods where continuous testing is vital. However, safety critical software requires strict measures and testing, and unlike formal methods cannot guarantee the absence of errors. Continuous formal verification [63] is a viable solution in this regard. Though there is a need for a significant amount of research to adopt a continuous formal verification process for automotive SDLC, insights from this article can be used to scale active learning to obtain formal models for safety-critical software development.

9.2. Practical Challenges

This section discusses practical challenges encountered in the course of the study.

9.2.1. 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 the application of automata learning in different scenarios over the years [29,48,49]. A major aspect of the active learning process is to establish a proper interface between the learner and the SUL. In this article, the interface is achieved through MATLAB-Java integration using the MATLAB Engine API for Java [54] as described in Section 5.2. A challenge is to establish an appropriate abstraction such that the learner can obtain necessary information about the alphabet to actively interact with the LSM. In this study, all external dependencies were abstracted such that the learner can easily interact with the SUL. However, 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. The effort needed to design and implement application specific learning setups can be reduced by creating test-drivers [48] in the form of standalone libraries, and/or automatically constructing abstractions [27] for seamless integration between the SUL and the learner.

9.2.2. Efficiency of the Learning Algorithms

The L^* algorithm in both the tools (MIDES and LearnLib) failed to learn a complete model of the LSM. From Section 7, it is evident that the number of membership and equivalence queries affect the efficiency of the L^* algorithm. Optimization by exploiting knowledge about the decision logic of the LSM was necessary to successfully learn a model of the LSM. While this corresponds to similar observations in other applications [29,55], it highlights the limitations in the practical applicability of language-based learning to autonomous driving software.

Though the MPL successfully learned a model of the LSM, more empirical case studies are needed to explore the limits of state-based learning in this context. The MPL is specifically developed to learn a modular system with several interacting automata. The main benefit here is the reduction in search space achieved by exploiting the structure of the SUL. Unfortunately, due to the structure of the LSM, a monolithic model had to be learned. However, the modular approach could potentially be helpful in tackling the complexity that arises in learning larger systems.

9.2.3. States vs. Events

Both 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; a small alphabet leads to a 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. 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, however, and both algorithms failed to learn a model. 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 improved by leveraging this trade-off when abstracting the code as seen in Section 7.2.

9.3. 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 theircorrectness. 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 adapt to the software reengineering lifecycle. Reengineering embedded automotive software is different from software reengineering in other domains due to unique challenges [64,65]. The active learning approach in this article can help identify unintended changes between different software implementations and also help to obtain high-level models from legacy systems, thereby aiding in the reengineering and the reverse engineering phase, respectively.

10. Conclusions

This article describes an application to interact with and learn formal models of MATLAB code. MATLAB/Simulink is currently a main engineering tool in the automotive industry, by automatically learning models of MATLAB code a significant step is taken towards the industrial adoption of formal methods. This is especially important for the development of safety-critical systems, like autonomous vehicles.

Using an active automata learning tool MIDES, which interacts with MATLAB, two different learning algorithms were applied to the code of a lane change module, the LSM, being developed for autonomous vehicles. One of these, an adaption of L^*, was unable to learn a model due to memory issues. The other, MPL, being a state-based method designed for learning a modular model, had more information about the target system, and learned a model in roughly one minute. To rule out possible implementation issues, another version of L^* from the open-source LearnLib toolbox was used to learn a model of the LSM. Even in this case, L^* failed to learn a model. Investigation of this negative outcome lead to alphabet reduction to improve the performance of L^*.

The learned models were validated in four ways:

- The language minimization of the model learned by MPL is very similar to the original model of the LSM.
- Manual comparison of the learned models to a manually developed model of the LSM indicated close similarity.
- Simulating the learned automata in SUPREMICA and comparing to the simulation of the actual code in MATLAB/Simulink showed no obvious discrepancies.
- A known bug in the development code was found also in the learned models.

Though the validation of the learned models were performed informally, taken together, they make a strong argument for the benefits of active automata learning in an industrial setting within the automotive domain. Model validation is a well known problem within the active automata learning community [29,31] and in the future, we would like to investigate different formal/semi-formal methods to validate the learned models.

Learning a monolithic model is a bottleneck, as it scales badly. Learning modular models potentially allows us 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.

Existing learning frameworks, like Tomte [27], could potentially help in the learning process by providing more efficient ways to abstract and reduce the alphabet. Furthermore, learning richer structures (but with the same expressive power), like extended finite state machines [66], is an interesting topic for further research. In addition, to further corroborate our findings, we plan to study several other software components of an autonomous vehicle, using the generic interface discussed in this article.

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

Author Contributions: Conceptualization, Y.S. and A.F.; methodology, Y.S. and A.F.; validation, Y.S., A.F. and M.F.; formal analysis, Y.S. and A.F; writing—original draft preparation, Y.S. and A.F.; writing—review and editing, W.A. and M.F.; supervision, W.A. and M.F.; project administration and article review, G.P. All authors read and agreed to the published version of the manuscript.

Funding: This research was supported by FFI-VINNOVA, under grant number 2017-05519, *Automatically Assessing Correctness of Autonomous Vehicles—Auto-CAV*, and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors thank Oskar Begic Johansson, Felix Millqvist, and Jakob Vinkås for their experiments with LearnLib and the work on alphabet reduction presented in Section 7.2.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Litman, T. *Autonomous Vehicle Implementation Predictions*; Victoria Transport Policy Institute: Victoria, BC, Canada, 2020.
2. Koopman, P.; Wagner, M. Autonomous vehicle safety: An interdisciplinary challenge. *IEEE Intell. Transp. Syst. Mag.* **2017**, *9*, 90–96. [CrossRef]
3. Broy, M.; Kruger, I.H.; Pretschner, A.; Salzmann, C. Engineering automotive software. *Proc. IEEE* **2007**, *95*, 356–373. [CrossRef]
4. Broy, M. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th International Conference on Software engineering, Shanghai, China, 20–28 May 2006;* Association for Computing Machinery: New York, NY, USA, 2006; pp. 33–42. [CrossRef]
5. Liebel, G.; Marko, N.; Tichy, M.; Leitner, A.; Hansson, J. Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice. *Softw. Syst. Model.* **2018**, *17*, 91–113. [CrossRef]
6. Charfi Smaoui, A.; Liu, F.; Mraidha, C. A Model Based System Engineering Methodology for an Autonomous Driving System Design. In Proceedings of the 25th ITS World Congress, Copenhagen, Denmark, 17–21 September 2018; HAL: Copenhagen, Denmark, 2018.
7. Struss, P.; Price, C. Model-based systems in the automotive industry. *AI Mag.* **2003**, *24*, 17.
8. Di Sandro, A.; Kokaly, S.; Salay, R.; Chechik, M. Querying Automotive System Models and Safety Artifacts: Tool Support and Case Study. *J. Automot. Softw. Eng.* **2020**, *1*, 34–50. [CrossRef]
9. The MathWorks Inc. MATLAB. Available online: https://se.mathworks.com/products/matlab.html (accessed on 17 February 2020).
10. Friedman, J. MATLAB/Simulink for automotive systems design. In Proceedings of the Design Automation & Test in Europe Conference, Munich, Germany, 6–10 March 2006 ; Volume 1, pp. 1–2.
11. Utting, M.; Legeard, B. *Practical Model-Based Testing*; Morgan Kaufmann: San Francisco, CA, USA, 2007.
12. Altinger, H.; Wotawa, F.; Schurius, M. 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, JAMAICA 2014, San Jose, CA, USA, 21 July 2014; Association for Computing Machinery: New York, NY, USA, 2014; pp. 1–6. [CrossRef]
13. Kalra, N.; Paddock, S.M. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transp. Res. Part A Policy Pract.* **2016**, *94*, 182–193. [CrossRef]
14. Luckcuck, M.; Farrell, M.; Dennis, L.A.; Dixon, C.; Fisher, M. Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.* **2019**, *52*, 1–41. [CrossRef]
15. Guiochet, J.; Machin, M.; Waeselynck, H. Safety-critical advanced robots: A survey. *Robot. Auton. Syst.* **2017**, *94*, 43–52. [CrossRef]
16. The MathWorks Inc. Simulink Design Verifier. Available online: https://se.mathworks.com/products/simulink-design-verifier.html (accessed on 17 February 2020).

17. The MathWorks Inc. Polyspace Products. Available online: https://se.mathworks.com/products/polyspace.html (accessed on 17 February 2020).
18. Leitner-Fischer, F.; Leue, S. Simulink Design Verifier vs. SPIN: A Comparative Case Study. 2008. Available online: http://kops.uni-konstanz.de/handle/123456789/21292 (accessed on 17 February 2020).
19. Schürenberg, M. Scalability Analysis of the Simulink Design Verifier on an Avionic System. Bachelor Thesis, Hamburg University of Technology, Hamburg, Germany, 2012.
20. Selvaraj, Y.; Ahrendt, W.; Fabian, M. Verification of Decision Making Software in an Autonomous Vehicle: An Industrial Case Study. In *Formal Methods for Industrial Critical Systems*; Springer: Cham, Switzerland, 2019; pp. 143–159.
21. Mashkoor, A.; Kossak, F.; Egyed, A. Evaluating the suitability of state-based formal methods for industrial deployment. *Softw. Pract. Exp.* **2018**, *48*, 2350–2379. [CrossRef]
22. Baier, C.; Katoen, J.P. *Principles of Model Checking*; MIT Press: Cambridge, MA, USA, 2008.
23. Ramadge, P.J.; Wonham, W.M. The control of discrete event systems. *Proc. IEEE* **1989**, *77*, 81–98. [CrossRef]
24. Liu, X.; Yang, H.; Zedan, H. 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, DC, USA, 13–15 August 1997; pp. 409–414.
25. Angluin, D. Learning regular sets from queries and counterexamples. *Inf. Comput.* **1987**, *75*, 87–106. [CrossRef]
26. Steffen, B.; Howar, F.; Merten, M. Introduction to active automata learning from a practical perspective. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 256–296.
27. Aarts, F. Tomte: Bridging the Gap between Active Learning and Real-World Systems Title of the Work. Ph.D. Thesis, Radboud University, Nijmegen, The Netherlands, 2014.
28. Cassel, S.; Howar, F.; Jonsson, B.; Steffen, B. Active learning for extended finite state machines. *Form. Asp. Comput.* **2016**, *28*, 233–263. [CrossRef]
29. Howar, F.; Steffen, B. Active automata learning in practice. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*; Springer: Cham, Switzerland, 2018; pp. 123–148.
30. Farooqui, A.; Hagebring, F.; Fabian, M. Active Learning of Modular Plant Models. *IFAC-PapersOnLine* **2020**, *53*, 296–302. [CrossRef]
31. de la Higuera, C. *Grammatical Inference: Learning Automata and Grammars*; Cambridge University Press: New York, NY, USA, 2010.
32. Isberner, M.; Howar, F.; Steffen, B. The open-source LearnLib: A Framework for Active Automata Learning. In Proceedings of the International Conference on Computer Aided Verification; Springer: Cham, Switzerland, 2015; pp. 487–495.
33. Cassandras, C.G.; Lafortune, S. *Introduction to Discrete Event Systems*; Springer: New York, NY, USA, 2009.
34. Selvaraj, Y.; Farooqui, A.; Panahandeh, G.; Fabian, M. Automatically Learning Formal Models: An Industrial Case from Autonomous Driving Development. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '20, Virtual Event Canada, 16–23 October 2020; Association for Computing Machinery: New York, NY, USA, 2020. [CrossRef]
35. Farooqui, A.; Hagebring, F.; Fabian, M. MIDES: A Tool for Supervisor Synthesis via Active Learning. In Proceedings of the 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE), Lyon, France, 23–27 August 2021; pp. 792–797. [CrossRef]
36. Farooqui, A.; Fabian, M. Synthesis of Supervisors for Unknown Plant Models Using Active Learning. In Proceedings of the 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE),Vancouver, BC, Canada, 22–26 August 2019; pp. 502–508.
37. Corbett, J.C.; Dwyer, M.B.; Hatcliff, J.; Laubach, S.; Pasareanu, C.S.; Zheng, H. 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, 4–11 June 2000; pp. 439–448.
38. Holzmann, G.J. From code to models. In Proceedings of the Second International Conference on Application of Concurrency to System Design, Newcastle upon Tyne, UK, 25–29 June 2001; pp. 3–10.
39. Holzmann, G.J.; Smith, M.H. 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, 16–22 May 1999; pp. 597–607.
40. Reicherdt, R.; Glesner, S. Formal verification of discrete-time MATLAB/Simulink models using Boogie. In Proceedings of the International Conference on Software Engineering and Formal Methods, Grenoble, France, 1–5 September 2014; Springer: Cham, Switzerland, 2014; pp. 190–204.
41. de Moura, L.; Bjørner, N. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*; Ramakrishnan, C.R., Rehof, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
42. Araiza-Illan, D.; Eder, K.; Richards, A. Formal verification of control systems' properties with theorem proving. In Proceedings of the 2014 UKACC International Conference on Control (CONTROL), Loughborough, UK, 9–11 July 2014; pp. 244–249.
43. Fang, H.; Guo, J.; Zhu, H.; Shi, J. Formal verification and simulation: Co-verification for subway control systems. In Proceedings of the 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering, Beijing, China, 4–6 July 2012; pp. 145–152.

44. Jonsson, B. Learning of Automata Models Extended with Data. In *Formal Methods for Eternal Networked Software Systems, Proceedings of the 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, 13–18 June 2011. Advanced Lectures*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 327–349. [CrossRef]
45. Aarts, F.; Schmaltz, J.; Vaandrager, F. Inference and Abstraction of the Biometric Passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*; Margaria, T., Steffen, B., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 673–686.
46. Aarts, F.; De Ruiter, J.; Poll, E. Formal models of bank cards for free. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 18–22 March 2013; pp. 461–468.
47. Smeenk, W.; Moerman, J.; Vaandrager, F.; Jansen, D.N. Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering*; Butler, M., Conchon, S., Zaïdi, F., Eds.; Springer: Cham, Switzerland, 2015; pp. 67–83.
48. Merten, M.; Isberner, M.; Howar, F.; Steffen, B.; Margaria, T. Automated learning setups in automata learning. In Proceedings of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Heraklion, Greece, 15–18 October 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 591–607.
49. Merten, M. Active Automata Learning for Real Life Applications. Ph.D Thesis, TU Dortmund University, Dortmund, Germany, 2013.
50. Kunze, S.; Mostowski, W.; Mousavi, M.R.; Varshosaz, M. Generation of failure models through automata learning. In Proceedings of the 2016 Workshop on Automotive Systems/Software Architectures (WASA), Venice, Italy, 5–8 April 2016; pp. 22–25.
51. Meinke, K.; Sindhu, M.A. LBTest: A learning-based testing tool for reactive systems. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, 18–20 March 2013; pp. 447–454.
52. Zhang, H.; Feng, L.; Li, Z. A Learning-Based Synthesis Approach to the Supremal Nonblocking Supervisor of Discrete-Event Systems. *IEEE Trans. Autom. Control* **2018**, *63*, 3345–3360. [CrossRef]
53. Chow, T. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* **1978**, *4*, 178–187. [CrossRef]
54. The MathWorks Inc. Java Engine API Summary. Available online: https://se.mathworks.com/help/matlab/matlab_external/get-started-with-matlab-engine-api-for-java.html (accessed on 17 February 2020).
55. Berg, T. Regular Inference for Reactive Systems. Ph.D. Thesis, Uppsala University, Uppsala, Sweden, 2006.
56. Czerny, M.X. Learning-Based Software Testing: Evaluation of Angluin's L* Algorithm and Adaptations in Practice. Bachelor Thesis, Karlsruhe Institute of Technology, Department of Informatics Institute for Theoretical Computer Science, Karlsruhe, Germany, 2014.
57. Zita, A.; Mohajerani, S.; Fabian, M. Application of formal verification to the lane change module of an autonomous vehicle. In Proceedings of the 2017 13th IEEE Conference on Automation Science and Engineering (CASE), Xi'an, China, 20–23 August 2017; pp. 932–937.
58. Malik, R.; Akesson, K.; Flordal, H.; Fabian, M. Supremica–An Efficient Tool for Large-Scale Discrete Event Systems. *IFAC-PapersOnLine* **2017**, *50*, 5794–5799. [CrossRef]
59. Rausch, A.; Brox, O.; Grewe, A.; Ibe, M.; Jauns-Seyfried, S.; Knieke, C.; Körner, M.; Küpper, S.; Mauritz, M.; Peters, H.; .; Strasser, A.; Vogel, M.; Weiss, N. Managed and Continuous Evolution of Dependable Automotive Software Systems. In Proceedings of the 10th Symposium on Automotive Powertrain Control Systems, 7 December 2014; Cramer: Braunschweig, Germany, 2014; pp. 15–51.
60. Patil, M.; Annamaneni, S. Model Based System Engineering (MBSE) for Accelerating Software Development Cycle. Technical Report, L&T Technology Services White Paper. 2015. Available online: https://www.ltts.com/sites/default/files/whitepapers/2017-07/wp-model-based-sys-engg.pdf (accessed on 27 December 2021).
61. Kubíček, K.; Čech, M.; Škach, J. Continuous enhancement in model-based software development and recent trends. In Proceedings of the 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 10–13 September Zaragoza, Spain, 2019; pp. 71–78.
62. Cie, K.M. *Agile in Automotive—State of Practice 2015*; Study; Kugler Maag Cie: Kornwestheim , Germany, 2015; p. 58.
63. Monteiro, F.R.; Gadelha, M.Y.R.; Cordeiro, L.C. Boost the Impact of Continuous Formal Verification in Industry. *arXiv* **2019**, arXiv:1904.06152.
64. Thums, A.; Quante, J. Reengineering embedded automotive software. In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 493–502.
65. Schulte-Coerne, V.; Thums, A.; Quante, J. Challenges in reengineering automotive software. In Proceedings of the 2009 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 24–27 March 2009; pp. 315–316.
66. Malik, R.; Fabian, M.; Åkesson, K. Modelling Large-Scale Discrete-Event Systems Using Modules, Aliases, and Extended Finite-State Automata. *IFAC Proc. Vol.* **2011**, *44*, 7000–7005. [CrossRef]