DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Automatically Learning Register Automata from MATLAB Code

A case study in autonomous driving

DEI ROSSI MARCO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Automatically Learning Register Automata from MATLAB Code

A case study in autonomous driving

MARCO DEI ROSSI

Master's Programme, ICT Innovation, 120 credits
Date: November 22, 2021

Examiner: Karl Meinke
School of Electrical Engineering and Computer Science
Host company: Zenseact AB
Swedish title: Automatiskt Learning Register Automata från MATLAB Code
Swedish subtitle: En fallstudie i autonom körning

Abstract

The successful verification of the behaviour of an Autonomous driving (AD) vehicle is fundamental for the commercialization of this new technology. Formal verification can be used to exhaustively verify the correctness of a system, but it requires a formal model to do so. An exact mathematical definition of the model description of the system is required. Manually defining a model is daunting, error-prone, and intractable for large systems.

Automata learning is a branch of Machine learning (ML) that automatically creates a formal model by dynamically interacting with the system. In prior research, automata learning algorithms have been shown to learn a formal model from AD software implemented in MATLAB. However, practical challenges were highlighted in addressing the state-space explosion problem and in obtaining suitable abstractions to deal with large systems.

To obtain valuable insights to scale up automata learning for industrial use, this thesis investigates the topic of automatically learning an extended finite automata formalism, called register automata. The *SL** algorithm, an extension of the *L** algorithm, is used to learn a register automata model of AD software from MATLAB code. The new algorithm creates a register automaton that manages to deal with the data dependencies intrinsically created from the case study between the input variables. Some approximations to the original model were made to obtain the desired solution. The obtained results are presented, from establishing the learning interface, validation of the interface, and from learning the case study. Evidence has been shown that similar problems to those highlighted for DFA learning are encountered. Future works have been discussed to address the same topic and to improve the proposed methodology.

Keywords

Automata learning, SL*, Register automata, RALib, Lateral State Manager

Sammanfattning

Den framgångsrika verifieringen av beteendet hos ett autonomt körande fordon är grundläggande för kommersialiseringen av denna innovativa teknik. Formella metoder kan förutses som den sista tekniken för att uppnå detta mål. Den exakta matematiska definitionen av modellbeskrivningen för systemet behövs. Tidigare verk kunde inte skapa en metod för att automatiskt beskriva beteendemodellen för SUL. Nuvarande *Lateral State Manager* varierar komplexiteten som förhindrade användningen av ramverk som LearnLib eller den normala L * -algoritmen. Den stora mängden ingångar och den exponentiella tillväxten av den observerbara tabellen är några av orsakerna som gör att standardalgoritmerna misslyckas. Tidigare löstes problemet delvis och introducerade viss abstraktion till SUL.

I denna avhandling föreslår vi en metod baserad på en innovativ algoritm: *SL* *. Den är utvecklad tack vare förlängningen av LearnLib-ramverket *RALib*. Den nya algoritmen skapar en registerautomat som klarar av att hantera databeroendet som skapats ur studiens fall. För att få en lösning inom acceptabel tid gjorde vi en ungefärlig tillnärmning till den ursprungliga modellen som vi tror inte påverkar det slutliga resultatet och skapar en modell som kan användas för de ändamål den var tänkt för.

Nyckelord

Automata learning, SL*, Register automata, RALib, LSM

Sommario

La corretta verifica del comportamento di un veicolo a guida autonoma è fondamentale per la commercializzazione di questa tecnologia. Diversi metodi formali possono essere previsti come una possibile tecnica al fine di raggiungere parte di questo vasto obiettivo. Al fine di utilizzare tali metodi è necessaria l'esatta descrizione matematica del modello comportamentale del sistema. Precedenti studi non sono stati in grado di creare automaticamente un metodo per descrivere il comportamento del sistema sotto esame. Il *Lateral State Manager* presenta varie complessità che hanno impedito l'uso di framework come LearnLib o dell'algoritmo L^*. La grande quantità di input e la crescita esponenziale della tabella delle osservazioni sono alcune delle principali cause che portano i comuni algoritmi al fallimento. In precedenza il problema è stato parzialmente risolto introducendo alcune astrazioni al caso di studio.

In questa dissertazione proponiamo un metodo basato su un algoritmo innovativo: *SL **. Lo stesso è sviluppato grazie all'estensione del framework LearnLib, chiamato *RALib*. Il nuovo algoritmo crea degli automi basati sui registri che riescono a gestire la dipendenza tra i dati intrinsecamente creati dal caso di studio. Per ottenere una soluzione in tempi accettabili si sono rese necessarie alcune approssimazioni del modello originale che riteniamo non influiscano sul risultato finale, creando un modello che possa essere utilizzato per gli scopi per cui è stato pensato.

Parole chiave

Automata learning, SL*, Register automata, RALib, LSM

Acknowledgments

Un pomeriggio di Novembre te ne sei andata. Sei stata il mio "grande amore" per tanti anni e non proverò mai le stesse cose per nessun altro. Tu c'eri, ci sei e ci sarai sempre, mi guarderai sempre dall'alto e mi proteggerai come hai sempre fatto quando mi eri vicina. Vivrò sempre con il rimpianto di non esserci stato nel momento del tuo bisogno, quando te ne sei andata, magari facendoti pensare che mi fossi dimenticato di te.
Questo è successo quando io ero in Svezia, per studiare, per prendere questa tesi. Quindi l'unica cosa che posso fare per provare a colmare questo vuoto è almeno dedicarti questa mia tesi, anche se non sarà mai abbastanza.

Sarai sempre nel mio cuore!! Ti voglio bene

A mia nonna
in memoriam

I would like to thank Yuvaraj Selvaraj, Ashfaq Farooqui and Karl Meinke for believing in me and supporting every day in every problem.

Thanks to Paul Fiterau for being my *Virglio* in understanding the library and helping me with the code.

Thanks to my group of friends in Stockholm that have always being there, for laughing and crying, for pushing me outside of my comfort zone, for all the lunches and dinners.

Thanks to all the friends I left behind, looking for something more, for always supporting me and for always being there when I came back.

Thanks to my roommate that stood me especially during the morning while I was talking.

Thanks to Cristina for the long days and nights in the library, listening to me and always helping me throughout all my doubts.

Thanks to my parents for supporting me, also during wrong decisions.

Thanks to all of you, because if today I am the kind of person I am, despite some bad features I have, it is also thanks to you, and I am proud of it.

Stockholm, November 2021
Marco DEI ROSSI

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

AD Autonomous driving

ADT Adaptive discrimination tree

API Application programming interface

CO-CPS Cooperative open cyber-physical systems-of-systems

DFA Deterministic finite automata

DHC Direct hypothesis construction

DRA Deterministic register automata

DT Deterministic timed

ECU Electronic control units

EFA Extended finite automata

FA Finite automata

FIFO First in first out

HAIM Heuristic Autonomous Intersection Management

KTH Kungliga Tekniska Högskolan

LBT Learning-based testing

LSM Lateral state manager

MAT Minimal adequate teacher

ML Machine learning

NL* Non deterministic L*

RA Register automata

RTSP Real Time Specification Pattern System

SAE Society of Automotive Engineers

SDT Symbolic Decision Tree

SL* Symbolic L*

SMC Statistical Model Checking

SPS Specification pattern system

SQA Software quality assurance

STA Spatio temporal automata

SUL System under learning

Chapter 1

Introduction

In the recent past, both industry and academia spent enormous efforts to introduce autonomous vehicles in the market. AD has the potential to benefit all road users in many ways, such as reducing the risk of accidents, speeding up traffic, cutting down the CO_2 emission, and many others. Perhaps the most important benefit is the possibility of AD to reduce human errors in the driving process. However, AD is a highly safety-critical task and appropriate measures must be taken to ensure their intended behaviour is safe and correct [2].

AD can be defined as one of the most significant challenges for Software quality assurance (SQA). Many safety critical tasks have to be brought to completion in very short amount of time (20-100 [msec]) varying from activation of actuators to decision making process [3].

Six levels of driving automation are defined by the Society of Automotive Engineers (SAE) in the J3016 standard, as published by the SAE in [4]. Levels from 0 to 2 implement driver support features, where the driver is still responsible for the decision-making during the driving process. Level 0 presents very little help for the driver and is limited to providing only warnings or momentary assistance. Level 1 provides steering or variation of velocity support. Level 2 is the last driver support feature that provides steering and braking/accelerating support. Levels from 3 to 5 are more focused on AD features, where the human driver is not responsible for driving activities when AD features are engaged. In Levels 3 and 4, the vehicle can independently operate under specific conditions. In Level 5, which is the highest level of automation defined, the vehicle can drive in all conditions without any supervision [5, 6].

Although AD features have several potential benefits, challenges exist in their successful deployment in the market. Problems such as perception, prediction, and the reliability of the AD software, that were considered to be more trivial in the idealization phase, are yet to be completely solved. As technology development is getting progressively harder, the development times are increasing [7]. In this regard, one of the most important challenges that slowed down the rapid growth of AD is the safety argument. AD features are safety-critical and therefore require rigorous methods to show that they are safe. The decision-making algorithms used need to be verified to provide proof of the correctness of their results, as incorrect algorithms could bring to catastrophic events.

The design and verification process for AD features is challenging in many ways. AD vehicles operate in a incredibly complex environment interacting with both fixed and moving objects. Sensors do not always work in optimal conditions and this further makes the decision making process challenging. The decision-making should be in real-time, safe, and not posing a threat to all participants of the environment at all times [6]. Determining if these systems behave as intended is an important task, albeit a difficult one. Safety standards and laws were issued to regulate the market and the safety of direct and indirect users. The most relevant is ISO 26262 that states the functional safety standard [8].

1.1 Testing and simulation

Various methods are commonly used to verify the correctness of AD software. Nowadays a popularly used method to provide some validation insights, which is also time-consuming, is testing on real vehicles. Testing is limited since it cannot prove the full correctness of the software, but can only fix errors when encountered. As stated in [9], it could be the case that more than hundreds of thousands of miles are needed to demonstrate safety, and assuming the testing is made properly. Testing in a real environment is intrinsically slow due to the physical laws that govern every action [10].

In addition to testing in a real environment, techniques such as computer simulations can be used to simulate complex traffic scenarios to test the software. It is estimated that only 165 hours of simulation can substitute 10 years of testing in the real world [11]. This opens the door to new, faster, and cheaper ways of development. The usage of technologies such as ML can provide new innovative techniques to test the system [12].

Although testing and simulation are widely used, such methods cannot provide any guarantee on the absence of errors. Every possible scenario should be created either in a real or in a simulated environment and evaluated. This is a reason for the inefficiency of such methods. From a mere logical point of view, it is impossible to theorize all cases,and therefore an exhaustive proof of safety is infeasible. It is crucial to prove safety in every single case.

Other methods are less error-prone and can consequently are more reliable. A higher level of insurance can be retrieved using computing technologies, such as formal methods or formal verification [13].

Nowadays none of those methods has demonstrated to be a complete approach to solve the task. It is therefore possible to foresee, at least for the moment, a combined used of all the existing methodology to achieve an acceptable solution to solve this challenge [3].

1.2 Formal methods

Formal methods are techniques which use rigorous mathematical models to verify software and hardware systems [14]. Formal methods could potentially address the challenges of testing and simulation approaches discussed in the previous section. Formal methods can provide a more rigorous proof for the correctness of the software, and therefore can complement testing to gain confidence in the correctness of the designed systems. A big advantage of formal methods is that they help in identifying subtle bugs and the so-called reasoning errors which might be left unverified by testing. Hence, formal methods could be useful for reasoning about safety-critical systems like AD [15, 16]. Formal methods in system design can be used at different levels such as formal specification, formal verification, formal synthesis, along with others.

Formal verification is the process of mathematically proving whether a formal model, that is describing the system's behaviour, satisfies certain requirements, which are also formally described [17]. For example, model checking is a formal verification technique which systematically checks whether or not a formal property holds in a finite-state model of the system [18]. Formal verification provides possibilities to exhaustively verify the behaviour of the system against its specifications.

Techniques like model checking are model-based and require an accurate model defining the logical behaviour of the system. Manually defining a model is a daunting and error-prone task. Moreover, it is intractable for large systems. Assuming it is feasible to create it manually, a small change in the studied system will require a much bigger effort to recreate the new model. This assumption cannot be made for systems like autonomous driving as they are rapidly evolving in time [1]. ML algorithms to automatically learn a model which describes the behaviour of a system are currently being researched [12]. Automata learning is a branch of ML which proposes to create a model dynamically interacting with the system [12]. Various techniques and algorithms have been developed during the past years.

1.3 Research questions

This thesis starts from the work proposed in [1], where the feasibility of active automata learning algorithms for automotive industrial applications is demonstrated. As a proof of concept, in [1], two different learning algorithms are investigated to learn a formal model from an AD software and the results are presented. Furthermore, practical challenges in applying automata learning for industrial automotive software development have been highlighted. Since active automata learning algorithms learn a model by actively interacting with the System Under Learning (SUL), one of the main challenges highlighted in [1] is to establish a proper interface between learning algorithms and SUL, such that all necessary information is communicated to the learning algorithms. In addition, [1] highlighted the well known state-space explosion issue as an important practical problem and observed that the efficiency of the learning algorithms can be improved by exploiting the structure of the code for abstractions of states during the learning process.

In light of the identified challenges, [1] proposed future research in the evaluation of automata learning algorithms using existing learning frameworks like LearnLib [19], Tomte [20], and through the investigation of learning formal models with richer formalism, like Extended finite automata (EFA). Based on these preliminary results, this thesis starts with the hypotheses that it is possible to learn a Register automata (RA) model for the given SUL and that automatically learning RA models is scalable even for automotive software defined in MATLAB.

In this regard, the final aim of the thesis is to investigate the learning process of a given SUL's RA model by answering the following research questions:

- **RQ1** What are the main automata learning algorithms for automatically extracting a formal model?

- **RQ2** Is it possible to learn a RA model of a given SUL in MATLAB? Under which conditions?

- **RQ3** Which could be a good method to automatically learn an RA?

The scope of the thesis is to automatically learn a RA model of an AD system developed in MATLAB as its SUL. To do so, this thesis first analyses different learning algorithms and papers to understand which one could better fit the case study [19, 20, 21, 22, 23, 24, 25, 26]. Afterwards, the focus will be on Register automata (RA) theory and already implemented tools to achieve the final goal. Secondly, the thesis involves the implementation of a suitable interface so that the learning algorithm can both actively interact with and infer a RA model of the SUL. The implemented interface is validated by learning a benchmark example and then proceeds to learn a model of the AD software under study. The thesis presents results from the validation of the interface and the analysis of the learnt model.

The author is aware of possible rejection reasoning which can bring the project to a negative result. From [1] and [27], it is known that a model of the SUL exists and is possible to learn a finite-state model automatically. No absolute guarantee can be provided for the creation of a model using different techniques. The achievement of the solution is highly dependent on the SUL and the algorithm used to learn. The dimension of the SUL could be a reason for the non-construction of the model. We can suppose that some assumptions are needed to reduce the SUL and to produce the final model.

1.4 Structure of the thesis

The thesis is structured starting from an initial literature study to fully comprehend the fundamentals of the project (Chapter 2), with a focus on the specific algorithm chosen and the case of study (Chapter 3). Additionally, in the same chapter all necessary prerequisites that will lead to the construction of the project are described. A structured methodology (Chapter 4) is presented in order to fulfill the aim of the project. It has been developed to simultaneously test and verify the final algorithm. Then, the thesis proceeds with the practical description of the code and the intended behaviour. The learning setup used in the project is described in detail in (Chapter 5). All the developed experiments have been presented, together with their results and conclusion. We believe this will improve the repeatably of the project and provide a better understanding of the methodology described in (Chapter 6). The implemented abstractions on the SUL are also presented as fundamental points to understand the results. A final needed validation is described to define the correctness of the work and the possible application of future work (Chapter 7).

Chapter 2

Automata learning

A behavioural model of a software or a mechanical component is needed to study the correctness of its behavior with model checking techniques. A model is supposed to illustrate different aspects or properties depending on the level of extent of the study. Looking at a system as a black box, it is possible to observe the output of the system in response to an ordered sequence of inputs. With such an information and understanding whose inputs trigger a change in the state, a model can be created. The more accurate the test process the more complex and complete model is created. The work can be done manually for simple case studies or not extensive tests, but computer-based techniques are required for large systems.

With these motivations in mind, an introduction to automata learning is presented in this chapter. The scope of automata learning is to automatically infer models from a set of data [12]. General principles or rules are extracted from the data and used to build the model. Automata learning is used to achieve a model of the System under learning (SUL), as presented in Chapter 3. The following overview starts from a more general point of view, before reaching the description of the final chosen implementation. This will draw to the conclusion of a feasible algorithm to be used to infer the solution of our project.

To properly represent an automaton model, in whatever way it has been generated, a graph can be used [28]. This is represented by a finite set of nodes, called states, connected through directed labelled arrows which represent the transitions. The labels represent the events associated with the transitions.

In Figure 2.1, it is possible to recognise two separate states: *on* and *off*. Arrows represent possible transitions, either that can bring from one state to the other or that close cyclically on the same starting state. Transitions are

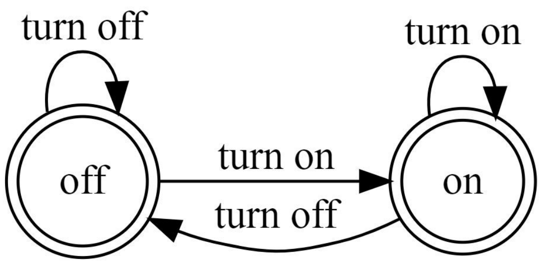


Figure 2.1: Example of a graph representing the model of a finite automaton (FA).

triggered from the two possible actions *turn on*, *turn off*. For example, if the graph starts from the *off* state and a *turn off* request is made, no change in state is performed. In case of a *turn on* statement, the state passes to the node *on*. The same conclusions can be drawn for every other state and transition. Later on, as it will be also used in graphs, intermediate states that help in the comprehension of the decision process in every transition. Intermediate states are needed to break the transition between two states. In this way, it is possible to represent both the function that triggers the transition and the output symbols produced by the same. Those are shown as double dotted circles. They are only drawn when they add useful information to the comprehension of the graph.

2.1 Strings, Alphabets and Formal Languages

The behaviour of systems can also be studied in other ways and formal language theory is one of them. Important definitions, such as alphabets, words, and languages, are needed to formalize in a unique way the method that is going to be used [29]. The differences between various algorithms are based on the usage of the elements that are proposed here below.

Systems can be described in terms of languages with their words and specific alphabets. An alphabet Σ is a finite and non empty set of actions. Formally, it is the list of all the atomical symbols that can be used by the system [30]. The alphabet is used to form words accepted by the system. A word is an ordered and finite set of symbols $\in \Sigma$. The empty string exists and is defined by the symbol ϵ. The set of all strings of certain length k in an alphabet Σ is denoted as Σ^k. The set of all strings of finite length over an alphabet Σ, including $\Sigma^0 = \epsilon$, is denoted by Σ^*. A formal language is said to be composed by the set of all and only words recognised by the system. A formal language $\mathcal{L} \subseteq \Sigma^*$ is a set of strings over Σ. A formal language is said to be regular when it can be defined using regular expressions that describe patterns over words [31].

Concatenation of strings is an important concept: If u and v are strings that $\in \mathcal{L}$, then uv is the string obtained by putting v immediately after u and it $\in \mathcal{L}$. This new string can be called x and is still recognised by the system. Using the same example in Figure 2.1 the accepted words that compose Σ are: *turn on* and *turn off*. Both of them are accepted from the system and belong to the alphabet. There are no other words in the alphabet characterizing the system. A possible concatenation can be *turn on-turn off* or other permutations of the two words. No limits on the length of the concatenated statement are imposed.

In the same way, a word composed as a concatenation of other words can be divided in two parts, called prefixes and suffixes. Formally, a string u is a prefix of a string x if there exists a string v such that $x = uv$. The part following the prefix is called suffix.

In the following sections, we will only focus on different types of automata models that found the methodology of this work. Deterministic finite automata (DFA) and then RA will be explained in detail thanks to the concepts defined above.

2.2 Deterministic Finite Automata (DFA)

Here is presented what is defined as the simplest automata model: DFA that are particular types of FA. A DFA is a finite state machine that recognises a regular language.

Definition 1 ***(DFA)*** *A deterministic finite automaton is a tuple* $\langle \Sigma;\ Q;\ q_0;\ F;\ T \rangle$*, where:*

- Σ *is a finite input alphabet;*
- *Q is a finite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $F \subseteq Q$ *is the set of final states; and*
- *T is a subset of* $Q \times \Sigma \times Q$*, such that for each* $q \in Q$*, each* $a \in \Sigma$ *and some* $q' \in Q$*, there is exactly one transition* $\langle q, \alpha, q' \rangle \in T$ *[30].*

DFA are characterized by a finite input alphabet Σ that returns values of a finite output alphabet Ω. These characteristics help in the convergence process of the learning algorithm to a solution. An automaton is said to be deterministic if, for every state, no two transitions from any given state are labeled with the same event [32]. One and only one initial state is present. The complexity of an automaton A could be defined by the size of A, i.e., the number of states in Q.

The algorithms for learning DFA use a black-box learning which could be very useful in projects implying dynamically changing or non-documented software. During learning, an DFA observes the behaviour of the studied system dynamically over time interacting with the SUL. Specific rules are defined to have a structured communication [33, 34].

Dynamically means that the learning algorithm performs queries in real time during the learning process to enlarge the knowledge about the system. Queries can be seen as an ordered set of events. Those events are sent as input to the studied system, that will provide the output of the system consequently to that request [35]. They are typically composed of a fixed prefix and a set of concatenated suffixes to be verified. The concatenated words are checked whether they belong to the system. Queries are created from an element called *learner* and answered by a *teacher*. Executing on the system the desired queries, it returns outputs that can provide an important information for the creation of the model. All information, queries and observations are saved and used to build the model. This is therefore clearly a supervised learning method. The right type and amount of queries should be used. Limited number of questions and answers can prevent the definition of the full model or not the desired one. Active automata learning algorithms are limited in accuracy by the number of queries [36]. The more queries, the higher the probability to have a complete model.

2.2.1 Myhill-Nerode theorem

As proposed before, DFA recognise a regular language. But a single regular language can be recognized by different versions of the same DFA. For this reason, minimization algorithms are needed to find the optimal, or rather the minimal model. The scope of a minimization algorithm is to determine the right number of states that are at the same time necessary and sufficient to fully represent the model corresponding to the regular language. The Nerode equivalence leads to the formalization of the Myhill-Nerode theorem [37, 38, 39].

Definition 2 ***(Nerode equivalence)*** *Let $\mathcal{L}$ be a language over the alphabet Σ, and let u and v be words from Σ^*. Then, u and v are Nerode-equivalent, denoted $u \equiv_{\mathcal{L}} v$, if for any word $w \in \Sigma^*$, we have that uw is in $\mathcal{L}$ if and only if vw is in $\mathcal{L}$ [30].*

Definition 3 ***(Myhill-Nerode theorem)*** *Let $\mathcal{L}$ be a regular language over an alphabet Σ, and $\equiv \mathcal{L}$ the induced Nerode equivalence on Σ^* with respect to $\mathcal{L}$.*

- *The language $\mathcal{L}$ is regular if and only if the number of equivalence classes of $\equiv \mathcal{L}$ is finite.*
- *The minimal deterministic finite automaton that recognises $\mathcal{L}$ has exactly as many states as the number of equivalence classes of $\equiv \mathcal{L}$ [30].*

L* is the most widely known minimal active learning algorithm and relies on the Nerode equivalence. L* algorithm studies the control flow of an SUL thanks to queries. The bigger the number of relevant queries, the more information can be gathered.

2.3 L* learning algorithm

In this dissertation, we mainly focus on active automata learning. The most famous algorithm to learn an unknown regular language is called *L** [34]. It will be presented to allow a fair comparison with its symbolic version *SL** presented later on. *L** aims to create a minimal DFA that recognises a regular language defined from a finite alphabet. A regular language is defined starting from fixed regular expressions that manages to define a unique pattern over all accepted words. For example, the regular expression $\{\alpha\alpha\beta^*\}$ describes all words that starts with two consecutive symbols α and continues with at least one β. We will use the regular language $\mathcal{L}_{\alpha\alpha\beta^*}$ as example to enable a better understanding for the reader of all the following definitions. $\mathcal{L}_{\alpha\alpha\beta^*}$ only accepts symbols $\{\alpha, \beta\}$. For example, strings as $\alpha\alpha$ and $\alpha\alpha\beta$ are words in $\mathcal{L}_{\alpha\alpha\beta*}$. Symbols starting with β, or having it as second element, are not accepted. Figure 2.2 represents the $\mathcal{L}_{\alpha\alpha\beta^*}$ DFA model.

For each node, the system proposes a transition corresponding to each symbol in the alphabet $\{\alpha, \beta\}$. The system starts from node q_0. If the transition follows the path $\{\alpha\alpha\beta\}$ the final output is accepted and terminates in the node q_2 shown with a double circle. If the input request does not respect the pattern defined by the regular expression, all transitions bring to a "sink" state q_3, where it is not possible to reach anymore the final state.

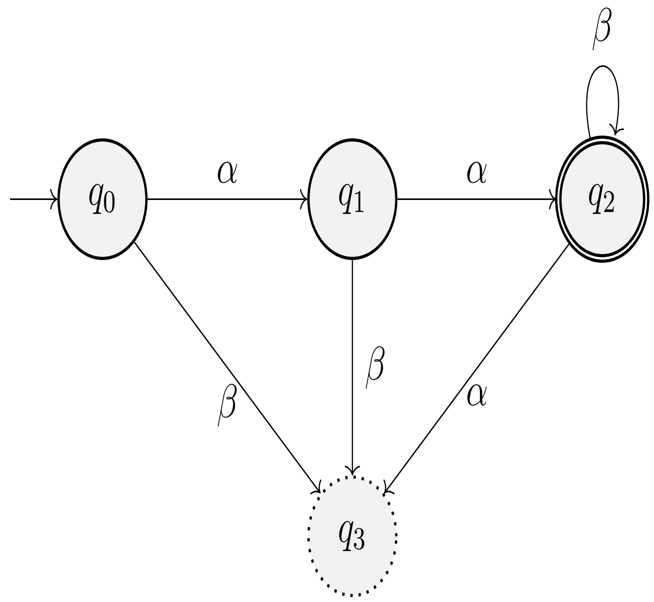


Figure 2.2: A DFA for $\{\alpha, \beta\}$ that recognizes the language $\mathcal{L}_{\alpha\alpha\beta^*}$.

L* has been developed to learn a general model from examples and queries. An agent called *teacher* provides examples to the learning algorithm, called the *learner*. A continuous interaction between the *learner* and the *teacher* is in place throughout the entire learning process to achieve the final scope. Examples need to be "helpful" to the construction of the model, so no useless information should be provided. In paper [34], a *learner* that efficiently learns an initially unknown regular set from a class of *teachers*, called Minimal adequate teacher (MAT), is presented.

A MAT answers correctly to the two types of queries posed by the *learner* regarding the unknown regular set. The first type of queries are *membership queries*. They consist of a string t of symbols that belong to the alphabet Σ and the *teacher* needs to answer yes or no depending on whether the string belongs to the set or not. The other type of queries are called *equivalence queries*. They are composed by a conjecture of the system. The answer to this type of queries could be yes in the case in which the hypothesis is equal to the studied system, it could be a string t provided as a counterexample highlighting the place where the conjecture differs from the system. Any *teacher* answers in the same way to membership queries since only one correct answer exists. However, in the case of an incorrect hypothesis, only a MAT can provide, if existing, any of the possible counterexamples.

Providing the first type of answer is undoubtedly easier than the second one. To answer the second question it requires the *teacher* to have a clear representation of the fully studied system. In the case in which this is not possible alternative methods should be adopted. Anyhow the answer is provided, this is not a trivial task and requires a lot of computational effort. No guarantees are provided on the conclusion of the learning algorithm. In [34] it is demonstrated that L^* correctly learns any regular set from any MAT in polynomial time in the number of states of the minimal DFA for the set and the maximum length of any counterexample provided by the Teacher. In [40] it is shown that passive learning of a DFA with a minimum number of states compatible with a given finite set of positive and negative examples can be an NP-hard problem. It means that even simple DFAs could result in computationally intractable problems. Additional information can be retrieved from the answer to membership queries to avoid the growth of the computational length. The L* algorithm tries to solve this problem with a structured methodology. As presented before, it manages to gain knowledge of the system thanks to membership and equivalence queries. L* collects information on the membership of strings and classifies their membership or not inside an observation table.

A set of prefixes is said to be prefix-closed if and only if every prefix of every member of the set is itself a member of the set. In the same way can be defined the concept of suffix-closed. The L* observation table is composed by rows and columns. Rows describe a non-empty finite prefix-closed set `S` of strings. Columns are nonempty finite suffix-closed set `E` of strings. The elements inside the table are defined as a function `T of` $\{+, -\}$ providing the membership of each element composed by the union of the corresponding prefix and suffix. The value is equal to 1 if the word composed by the prefix and the suffix is a member of the studied set. Suffixes are needed to distinguish prefixes depending on `T`. The observation table is denoted as `(S,E,T)`. Initially, the table is empty with no rows or columns filled. As the algorithm proceeds and membership queries are evaluated, the table starts growing. Formally, the observation table is defined as:

Definition 4 ***(Observation table)*** *An observation table for L* is a tuple* $O = (S, E, T)$ *where:*

- S *is a prefix-closed set of short prefixes;*
- E *is a set of suffixes; and*
- $T : (S \cup (S \cdot \Sigma))\ X\ E \mapsto \{+, -\}$ *is a labeling function [30].*

Prefixes are classified in two groups, depending on their behaviour `T(s,e)`. Graphically the table is vertically divided in two parts. The top part contains short prefixes, the bottom part contains extended prefixes. Every short prefix should have a different behaviour compared to any other short prefix. If a new studied prefix, originally presumed extended, shows a new behaviour `T(s,e)` that is not present in the upper part of the table, it is redefined as short prefix and promoted to the upper part.

Observation table					
Prefix	Suffixes				
	ϵ	α	$\alpha\alpha$	β	$\alpha\beta$
ϵ	-	-	+	-	-
α	-	+	-	-	+
$\alpha\alpha$	+	-	-	+	-
$\alpha\beta$	-	-	-	-	-
$\alpha\alpha\beta$	+	-	-	+	-
β	-	-	-	-	-
$\beta\beta$	-	-	-	-	-
$\beta\alpha$	-	-	-	-	-
$\alpha\alpha\alpha$	-	-	-	-	-

Table 2.1: Example of observation table for the language $\mathcal{L}_{\alpha\alpha\beta^*}$ created from L^* algorithm.

L* is based on the Nerode equivalence and Myhill-Nerode theorem, as presented in Definitions 2 and 3. Initially all words are assumed to be Nerode equivalent and no distinctions on suffixes are contemplated. Using the knowledge obtained from the *teacher*, a distinction can be made on suffixes. The observation table can introduce equivalence relations on prefixes relying on those concepts. Two prefixes are said to be equivalent if their behaviour with all the studied suffixes is the same. It is denoted as $\equiv_O$. Formally, two prefixes `s` and `s'` are equivalent (i.e., $s \equiv_O s'$), if `T(s,e) = T(s',e)` for all suffixes `e` of the table. An observation table is closed if all prefixes have an equivalent short prefix.

Definition 5 ***(Closed and consistent observation table)*** *An observation table is a tuple $O = (S, E, T)$ is closed if for any prefix $s' \in (S \cup (S \cdot\Sigma))$, there is a short prefix $s \in S$ such that $s \equiv_O s'$ [30]. An observation table is said to be consistent provided that whenever s' and s are element in S such that $s' \equiv_O s$, for all $\alpha \in \Sigma$, $s'\alpha \equiv_O s\alpha$ [34].*

The table is said to be complete when it is closed and consistent. Given a closed observation table, it is possible to construct a hypothesis for the DFA that describes the studied system.

Definition 6 ***(Constructing a finite automaton from an observation table)*** *A closed and consistent observation table* $O = (S, E, T)$ *can be used to construct a DFA* $A = \langle \Sigma, Q, q_0, F, T \rangle$*, in the following manner:*

- Σ *is a finite input alphabet;*
- *the set* Q *of states is obtained from the set* S *of short prefixes: each short prefix* $s \in S$ *represents a state* $q_s \in Q$*;*
- *the initial state is* $q_0 \in Q$ *is represented by the short prefix* ϵ*;*
- *the set* F *consists of the accepting states in* Q*. A state* $q_s \in Q$ *is accepting whenever* $T(s, \epsilon) = +$*, and rejecting (-) otherwise;*
- *the set* T *of transitions is extracted from the set of prefixes, so that for all* $s \in S$ *and* $\alpha \in \Sigma$*, we get* $\langle q_s, \alpha, q_{s'} \rangle \in T$ *where* s' *is the unique short prefix such that* $s\alpha \equiv_O s'$ *[30].*

The operation of the L* algorithm is usually separated into three phases that run cyclically: hypothesis construction, hypothesis validation, and counterexample processing. The SUL is a black box system, so the *learner* does not have any information on the structure of the SUL. Using membership queries, the *learner* gains knowledge of the system and builds the observation table in the same way as in the theory presented above. Queries are continuously performed until the table is closed. Promotions to short prefixes are performed when necessary. As soon as the table is closed, the algorithm constructs an hypothesis automaton following Definition 6.

Using the same communication paradigm as the one used for membership queries, the presumed model is sent to the *teacher* for an equivalence check. If the *teacher* has had a full knowledge of the system, the answer to this type of queries would have been trivial. Practically, the *teacher* is not always omniscient, so alternative methods need to be used. Random checks for equivalence can be tested and typically manage to produce an acceptable answer. More complex methods can be also implemented to provide the guarantee of a minimal solution [41, 42, 43]. The L^* algorithm is not going to be used for this project, so equivalence checking is not presented here in detail, since it will be better explained for the SL* algorithm.

Anyhow, the equivalence check is performed. If the model perfectly represents the system, it is accepted, and the goal accomplished. If the model and the system differ, a counterexample that highlights the error in the model is provided. A counterexample is a word where the *teacher* and the *learner* do not agree. The word is divided in a prefix and a suffix, where the prefix is already existing in the short prefix set. Processing this word will bring about the creation of a new suffix on the existing short prefix that creates a new state of the automaton. The suffix is added to the set of suffixes and all prefixes are reanalyzed in this new configuration. The final model produced for the benchmark example $\mathcal{L}_{\alpha\alpha\beta*}$ is the same as the one proposed in Figure 2.2:

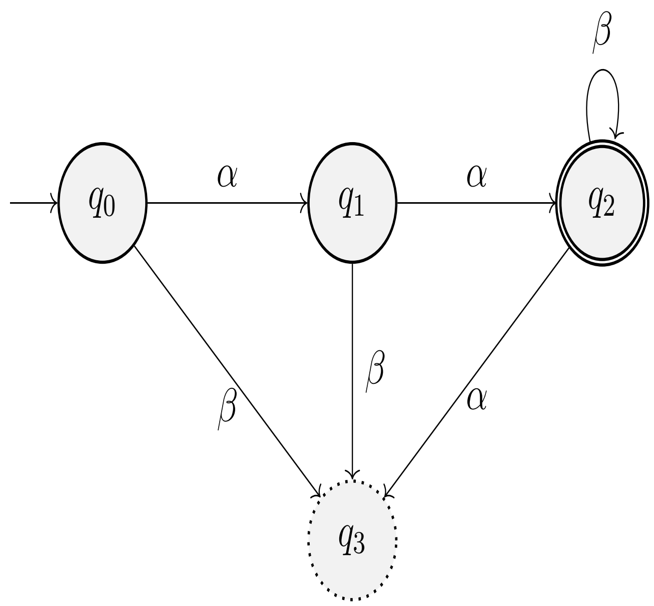


Figure 2.3: DFA model of the $\mathcal{L}_{\alpha\alpha\beta*}$ regular language created after the learning process.

Large number of queries could make the state space grow. The complexity of a system is nothing more than a large state space and a growing computational effort required for large systems. Applying automata learning algorithms to complex systems would be impractical and time consuming. Introducing some simplification in the learning process of the SUL could permit to represent the system with a richer formalism of DFA, such as RA. Thus, learning an RA instead of an DFA is more practical for real world data flow applications [44].

2.4 Register Automata (RA)

Not all systems can be modelled adequately and succinctly by finite state machines. This is mainly due to the size of the system and the state space explosion. Particularly, if the input alphabet belongs to an infinite domain, the system cannot be represented at all as a simple FA. Extended finite automata (EFA) can be seen as a possible solution to the previously presented problem [30]. Those types of automata focus their operation on relations between inputs rather than the effective value of the same. RAs are particular types of EFA and we believe they could provide a good solution to our research questions. Register will be used in the practical implementation of the learning setup. We will limit the presentation of RA only to the case of Deterministic register automata (DRA) and all needed definitions to properly understand the operation of the learning algorithm. As the names conveys, RA extend FA with storage units called registers.

Providing the reader with an example can help in the comprehension of the following definitions and later on the operation of the learning process. A First in first out (FIFO) queue system has been chosen as benchmark experiment because it presents most of the characteristics that can help in the explanation of an RA, like the usage of registers. Different levels of complexity of the FIFO have been implemented to better describe each feature of the proposed method.

A FIFO is a data structure that defines a clear logic for handling elements inside an array. The first entry in the system is also the first element processed. Here it is described as a practical system where elements can be inserted inside a fixed size array and given as output when requested. The array is bounded so that it can be described by an FA. It accepts two functions: `push` and `pop`. Function `push` is characterized by the value that needs to be saved in the system. `Pop` does not need specific arguments to fully characterise the desired request. To simplify the discussions only one of them, `push`, is going to be presented in detail. The other function is fairly self-explanatory and will be left to the reader for exploration.

This system can be fully described by an automata model and can be depicted with a set of locations, connections between them, and drawn using a graph similarly to DFA. Register automata are reminiscent of control flow graphs [24].

The classes of languages that can be recognised by DRA are referred to as regular data languages. The characteristic parameter of an RA is the usage of registers that can store important values. An RA allows us to assign values from large domains to registers and to compare them with various specific theories. In [24] a canonical representation of RA has been proposed with promising results. Canonical indicates, in this thesis, an RA that recognises the model with the minimum number of states. It is possible thanks to an adaptation of the Myhill-Nerode theorem. One of the major contributions of [24] is the definition of a unique representation for any language recognizable by an DRA.

The most important features that distinguishes RA from DFA are that an RA is based on the concept of data symbols and the usage of registers. These notions will immediately require a new definition of transition function.

Definition 7 ***(Data symbols)*** *By a data symbol we mean a string $\alpha(d)$ where α is an action accepted from the alphabet and d is the data value that can vary inside the same action. The words are defined in the alphabet, instead d belongs to the domain D. Data words are used to denote an ordered sequence of data symbols [45].*

This implies that the symbols accepted by the system are composed of an action α and a value `d` that can vary inside the same action. The domain `D` can be for example, the set of integer or natural numbers, so the data values `d` are numbers belonging to that set.

In the benchmark FIFO example, actions α are `pop` and `push` functions and data values are the `d` parameters in the `push` function. `Pop` does not present any data value. Data symbols $\alpha(d)$ are represented as:

- `push(d)`. This is the request for adding a value `d` to the system. The internal logic will automatically define the exact location of the element;

- `pop( )`. This request produces an output of the element in the system defined by the internal logic of the model.

The concatenation of two data words $\alpha(d)$ and $\alpha'(d')$ is denoted by $\alpha(d)\alpha'(d')$. Using this notation is common to define $\alpha(d)$ as prefix of the new constructed word and $\alpha'(d')$ as suffix. The set of actions α belonging to a data word $\alpha(d)$ are denoted as $Acts(\alpha(d))$. The sequence of data values is denoted by $Vals(\alpha(d))$. The sequence of actions in Σ^* (defined in Section 2.1) after the prefix, are referred as symbolic suffixes. We define V for a set of symbolic suffixes, and [[V]] for the set of words $\alpha'(d')$ with Acts($\alpha'(d')$) $\in$ V. Consequently, $\alpha(d)$[[V]] is the set of words of the form $\alpha(d)\alpha'(d')$ with Acts($\alpha'(d')$) $\in$ V. If V is a set of symbolic suffixes, then $\alpha^{-1}V$ is the set of subsequences of suffixes in V such that $\alpha'(d') = \alpha_1\alpha_2...\alpha_n$ is in V if and only if $\alpha''(d'') = \alpha_2...\alpha_n$ is in $\alpha^{-1}V$ [34].

In the benchmark example, an acceptable word for the system constructed as the concatenation of two others can be `(push(d)push(d')push(d″)pop())` (`d`, `d'` and `d″` can also have the same value). Practically, the system accepts three parameters `d`, `d'`, `d″` and saves them into the array and then outputs `d`. The word can be divided into two parts, e.g., prefix `(push(d) push(d'))` and suffix `(push(d″) pop())`. Prefix and suffix should not be mandatorily of the same length. Prefixes can be seen as a set of actions ($Acts(\alpha(d))$) and a set of data values ($Vals(\alpha(d))$). For the prefix `(push(d) push(d'))`, $Acts(\alpha(d))$ = `(push, push)` and $Vals(\alpha(d))$ = `(d, d')`. For the prefix, $Vals(\alpha(d))$ are fixed and uniquely define the concrete prefix studied. The set of symbolic suffix only present $Acts(\alpha'(d'))$ = `(push, pop)`. Data values are not fixed, permitting a full study of all possibilities after the prefix with the required action structure.

RAs distinguish words depending on the action and the relation between data values. Relations are mainly processed at a symbolic level, not looking at the effective value, but rather on the trend they have one in relation to the other. Different functions and different data values can bring the RA to separate states. The most intuitive way to represent this type of division is the usage of trees. It is possible to discriminate different output states starting from a parent node. Clearly, for the FIFO example, starting from the initial state q_0, the inputs `push(d)` and `pop( )` brings the RA to two separate states. Figure 2.4 shows the distinction between state q_1 and q_2. The standard FIFO system do not permit to differentiate the state depending on data values, e.g., `push(d)` and `push(d')` bring to the same state.

Theories are needed to define a unique rule to compare different data values. A theory is defined as a pair $\langle$ `D`, `R` $\rangle$, where `R` is the set of relations on the data domain `D`. RAs extend the finite set of states, used for DFA, with a set of registers used to store the data values. RAs give the possibility to compare data values belonging to the same set `D`, with both equality or inequality relations . The introduction of the concept of `R`-indistinguishable words is needed to formalize the equality and inequality comparison. Equality theory checks data values belonging to the controlled actions with equality operations. Using equality theory, it is only possible to check data values with the equivalence operator, not considering the incremental evolution of the same. Inequality theory, instead, manages to control the relation between data values with inequality operators as proposed in Definition 8.

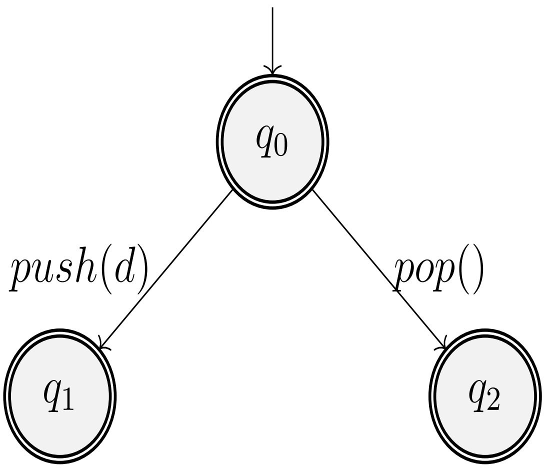


Figure 2.4: A graph for a FIFO system that accepts both $push(d)$ and $pop()$.

Definition 8 ***(R-indistinguishable data words)*** *Two data words $\alpha_1(d_1)...\alpha_n(d_n)$ and $\alpha'_1(d'_1)...\alpha'_n(d'_n)$ are said to be $\mathbb{R}$-indistinguishable if :*

- *they present the same sequence of actions. Formally $\alpha_1...\alpha_n = \alpha'_1...\alpha'_n$;*
- *R$(d_1, ..., d_j)$ if and only if R$(d'_1, ..., d'_j)$ whenever R $\in \mathbb{R}$ and i, ..., j are indices between 1 and n [30].*

Thanks to this definition, it is possible to determine *R*-indistinguishable data words and group them in only one symbol. This process depends on the theory that is applied to the study. Adapting the definition of language in Section 2.1, it is possible to say that a data language is composed by a set of words that follows the *R* relations. More specifically, if two data words are R-indistinguishable, they both belong or not in the data language.

Although both equality and inequality theories are going to be used in the learning process, they are not essential for the FIFO example because no distinctions in the data values are made. The FIFO example does not permit to properly show the operation of these definitions. A more trivial example will clarify those definitions and the possible usage of those theories:

- $\alpha(1)\alpha(1)\alpha(2)$ and $\alpha(3)\alpha(3)\alpha(3)$ are distinguishable words thanks to equality theory. Although the sequence of action α is the same, the data values are not following the same trend. The first words have the first data value equal to the second, but different from the third. The second word does not follow the same trend because all three parameters are equals. $(d_1 = d_2 \neq d_3)$ is different from $(d_1 = d_2 = d_3)$;

- $\alpha(2)\alpha(2)\alpha(1)$ and $\alpha(3)\alpha(3)\alpha(8)$ are indistinguishable with the equality theory since all the three actions and data values follow the same course $(d_1 = d_2 \neq d_3) \equiv (d'_1 = d'_2 \neq d'_3)$. For inequality theory, those two words are distinguishable because the trend of data symbols is not respected (the third data symbol is in one case less than the other two, in the other case is more). The data symbols of the first word are studied as $(d_1 = d_2 > d_3)$, instead for the second word is $(d'_1 = d'_2 < d'_3)$. $\alpha(1)\alpha(1)\alpha(2)$ is indistinguishable also with respect to the inequality theory when compared to $\alpha(3)\alpha(3)\alpha(8)$.

All data values of different words in the same relative position are compared with inequality functions. So $\alpha(1)\alpha(1)\alpha(2)$ and $\alpha(3)\alpha(3)\alpha(8)$ are still indistinguishable, as both respect the inequality between the first two and the last data value. In other words, in both cases the third unit is bigger than the other two elements.

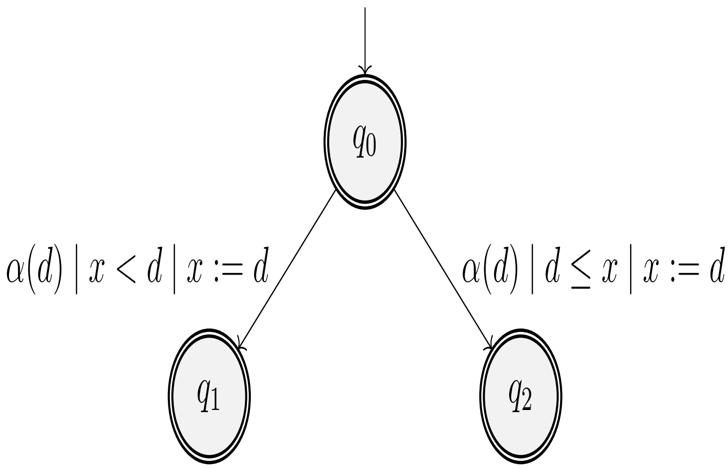


Figure 2.5: A general graph differentiating branches with inequality theory.

In the Figure 2.5, it is possible to clearly see that the same data symbol $\alpha(d)$ brings to different states depending on the value of the data parameter. The example has been developed following the inequality theory. Starting from state q_0 and with $\alpha(d)$ input function, if the data value is bigger than the value saved in the register x the final state is q_1. Instead, if $d \leq x$ the final state is q_2

In the case in which the system cannot be learned with equality or inequality theories, as in the FIFO example, RA could also provide a further check on the effective value of data values with constant parameters. Constants need to be provided before the learning process starts. This implementation will change the learning algorithm from a black box to a grey box process. Constants describing specific data values are meant to identify specific cases that distinguish different states depending on the data value that triggers the change. Those are useful in the case in which the data type domain is big or even unbounded, e.g., natural or integer numbers. All and only the data values that bring to a new state should be checked. All of them in order not to lose generality, no other irrelevant values should be tested to limit the explosion in dimension of the study. If more than one data value activates the same transition function, those should be merged consistently with Nerode equivalence. Constant values are defined to optimize the algorithm following the previously presented logic. The learning algorithm checks every function with all the constants, and for a single value different from every other constant. In this way, every final state is controlled.

The standard FIFO example does not contemplate the usage of constants. A minor change can be introduced in the description of the studied system to allow the usage of constant parameters. A distinction in the data values must be introduced. For example, a constraint can be imposed in order to reject the request when the input function is: $?push[2]$. In this case, if the system tries to evaluate a push action with the specific data value 2 into the system, the system return a statement ($!\_not\_ok[\ ]$). The data value equals to 2, it triggers a transition in the push function. Every other integer brings to a same state. In this case, 2 becomes a constant checked from the system in every evaluation. The studied functions are $?push[c1 == d1]$ and $?push[c1! = d1]$.

Each input function must be verified with a specific theory, such as equality or inequality theory. Constants can be defined and consequently controlled. The mathematical definition of conditions on data values is called a guard. If different data values in the same action produce different behaviours, a guard is needed which characterizes unequivocally the description of the transition. The scope of guards is to specify the condition the data value needs to have against the corresponding part saved in the register, accordingly with the specified theory. Registers always have a consistent dimension with respect to the compared data value. Guards usually consist of conjunctions, disjunctions, or negations of multiple conditions, depending on the dimension of the register.

Moreover, guards can differentiate the transitions using constants parameters and equality theory. Using the updated FIFO example it is clear that, if the data value is equal to constant `c`, a differentiation in the output state is produced. In Figure 2.6, it is possible to graphically understand the aforementioned situation.

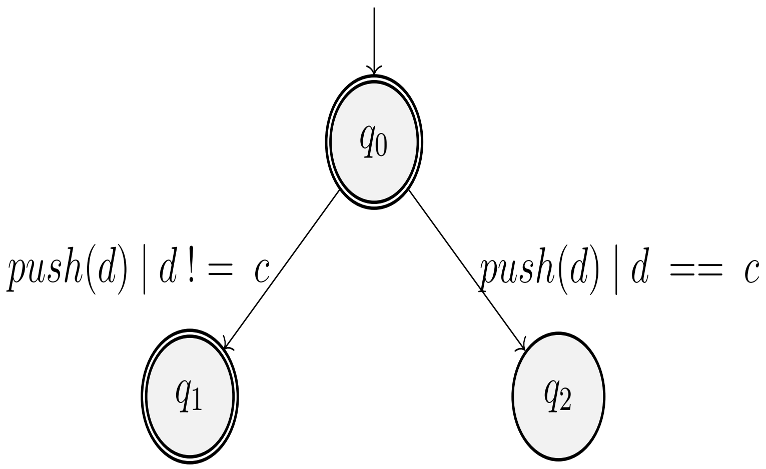


Figure 2.6: A SDT for a FIFO system with constraint that accepts both $push(d)$ when d is different from the constant parameter.

It is possible now to clearly understand why EFA and specifically RA can help in the creation of a model for system characterized by huge domain size. Instead of studying each case for every possible data value and data symbol, only the relevant cases are going to be studied. Guards permit to divide the infinite set of numbers characterizing the data type (integers, real, ...) into limited and much smaller sets and cases to be studied. Therefore, the number of studied cases is no longer infinite, but follows the following formula. It corresponds to the number of sets `m` as the number of variables that define the possible output states `n` plus the one that rejects all the other cases.

$$\infty \rightarrow \texttt{m=n+1}$$

Studying the full system considering all cases can bring to a non-converging solution. Analyzing a smaller or simplified case can help in the learning process, but a non-desirable solution can be produced. Innovative solutions, characteristic of SL*, should propose a much faster and precise convergence of the algorithm to the real model. We can therefore expect a boundary related to the instantaneous capabilities of the system. Guards are defined depending on the theory used.

Transition functions for RA are therefore uniquely defined by the previously presented components:

- $\alpha(d)$ is the data symbol with actions belonging to the set `U` and data values, usually from the natural number set. In the case of boolean variables, they could also be 0,1;
- the mathematical definition of guard (e.g., `x ≤ d`). As introduced in the RA section, guards check the relations between data values `d` and the registers `x`. It is possible to define guards using equality or inequality theories;
- `x := d` defines the update function of the register with the new data value number.

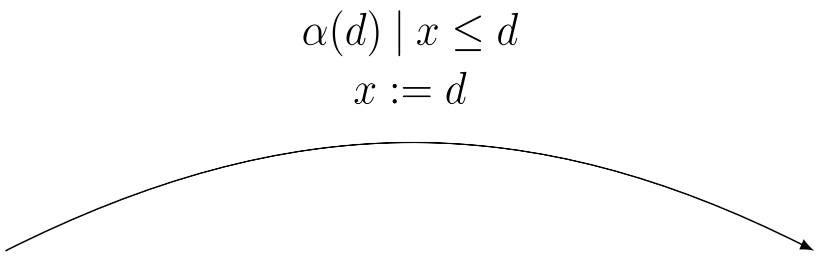


Figure 2.7: General definition of transition function.

A more mathematical definition of RA can be given to formalize the previously explained concepts.

Definition 9 ***(RA)*** *An RA is a tuple $\langle Q, q_0, F, \chi, \Gamma \rangle$, where:*

- *Q is a finite set of locations;*
- *$q_0 \in Q$ is the initial location;*
- *$F \subseteq Q$ is the set of final (accepting) locations;*
- *χ maps each location $q \in Q$ to a finite set $\chi(q)$ of registers; and*
- *Γ is a finite set of transitions, each of form $\langle q; \alpha(d); g; \pi; q' \rangle$ where:*
  - *$q, q' \in Q$ are respectively the source and target locations;*
  - *$\alpha(d)$ is a data symbol composed by an action α and a data value d;*

- *g is a guard over d and $\chi(q)$; and*
- *π is the assignment, a mapping from $\chi(q')$ to $\chi(q) \cup d$. Intuitively, the register $x' \in \chi(q')$ is assigned the value of $\pi(x')$, which is either the parameter p or some register $x \in \chi(q)$ [30].*

For graphical representation, trees have been adopted. It is now possible to introduce Symbolic Decision Tree (SDT)s since all necessary prerequisites have been explained. SDTs are defined to symbolically represent relations between data values in prefix and suffix. An SDT starts from a parent node that is corresponding to a prefix with a concrete(fixed) data value. Each child node represents a symbolic suffix, i.e., a sequence of symbols with uninstantiated data parameters (called parameterized symbols). A state is accepted if the suffix fully described by the corresponding transition function, brings to an acceptable location starting from the root location. The root location is depicted at the top of the graph. Locations are either accepting (double circles) or rejecting (single circle). Branches are characterized from transitions as described in Figure 2.7.

An SDT can be seen as a restricted part of an RA in the form of a tree. An SDT that recognises a regular set of data words manages to fully describe which words are accepted and which are rejected using relations between data values. It models the acceptance of a set of suffixes `V` after a specific prefix $\alpha(d)$.

Definition 10 ***(SDT)*** *An SDT is an RA T = $\langle Q; q_0; F; \chi; \Gamma \rangle$, where Q and Γ form a tree rooted at q_0 [34].*

The SDT is constructed by a tree oracle. To have a minimal automaton a canonical tree oracle is needed. It follows the descriptions imposed by the theory ⟨*D,R*⟩. To minimize the number of branches, transitions that brings to the same state need to be merged. Two SDTs that have the same structure are said to be isomorphic. A bijection function γ exists between the locations of two isomorphic SDTs and ensures that their locations, transitions, and registers are the same. The two SDTs in Figure 2.8 and 2.9 are isomorphic and can be therefore merged in only one SDT. An SDT can also be described with a couple `(u,V)` of a prefix `u` and a set of symbolic prefixes `V`.

Definition 11 ***((u,V)-tree)*** *For a data word `u` with `Vals(u)` $= d_1, ..., d_k$, and a set `V` of symbolic suffixes, a `(u, V)-tree` is an SDT `T`, which has runs over all data words `v` $\in$ `[[V]]`, and which satisfies the technical restriction that whenever $\langle q; \alpha(d); g; \pi; q' \rangle$ is the jth transition on some path from q_0, then for each $x_i \in \chi(q')$ we have either*

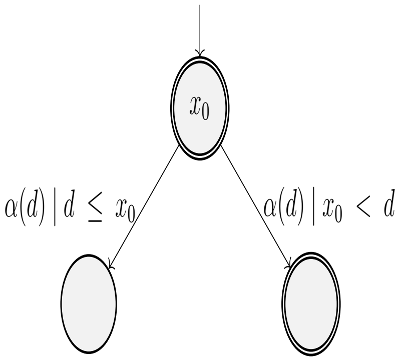


Figure 2.8: SDT on register x_0

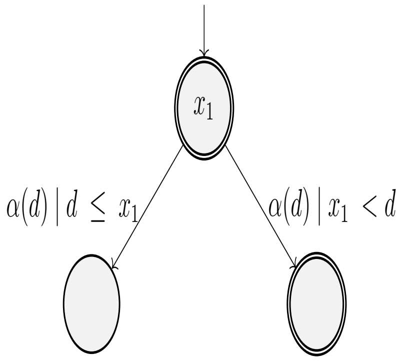


Figure 2.9: SDT on register x_1

1. $i < k + j$ *and* $\pi(x_i) = x_i$, *or*

2. $i = k + j$ *and* $\pi(x_i) = d$ *(recall that k is the length of u) [36].*

Intuitively, a `(u,V)-tree` is an SDT for a set `V` of symbolic suffixes after a unique prefix `u` and where, in any run over a data word `v` $\in$ `V`, the register x_i may contain only the value of the ith data value in `uv`. This restriction is introduced to process the comparison of two `(u,V)-trees` easier.

Definition 12 ***(Tree oracle)*** *Let ⟨D,R⟩ be a theory. A tree oracle for ⟨D,R⟩ is a function* `O`*, which for a language* $\mathcal{L}$*, data word* `u` *and a set* `V` *of symbolic suffixes returns a* `(u, V)-tree O(`$\mathcal{L}$`, u, V)`*, such that for any data word* `uv` $\in$ `u[[V]]` *it holds that v is accepted by* `O(`$\mathcal{L}$`, u, V)` *under* v_u *if and only if* `uv` $\in \mathcal{L}$*, and rejected if and only if* `uv` $\notin \mathcal{L}$ *[34].*

To construct a canonical RA, not any tree oracle can be used. A specific type of tree should be proposed to construct a canonical RA. Canonical tree oracle manages to satisfy all the needed features to construct the desired minimal model. The properties to be satisfied are:

- trees are constructed recursively;
- extending the set of symbolic suffixes, two previously inequivalent symbolic decision trees can not be made equivalent;
- whenever we extend the set of symbolic suffixes, the guards of the initial transitions either stay the same or are refined, while the set of registers is either the same or extended [34].

Different tree oracles have been proposed in [34] to perfectly fit the different theories that can be used from the learning algorithm, but all lead back to this previous definition. Extensions of the tree oracle, like the usage of constants, have also been implemented.

A practical example can help in the comprehension on how a SDT is generated by the canonical tree oracle. Let's consider two words `u = `α`(5)` α`(2)` and `v = `α`(8)` α`(4)` and the inequality theory that will enable the comparison with $\{=, <\}$ symbols. With our previous knowledge, we know that it is not possible to distinguish those words. It is possible to compare them at a symbolic level, focusing on relations between data values. The process is practically divided in three steps:

1. Each word is extended with a set of suffixes in the form $\alpha(d_3)$. Those are not chosen randomly but are all those that are relevant to the studied word. Those are one for each set that can be distinguished and studied by the chosen theory. For example, the suffixes to extend the word `u = `α`(5)` α`(2)` are:
   - $(\alpha(1)$ where $d_3 = 1 < d_2(= 2)$ && $d_3 < d_1(= 5)$;
   - $(\alpha(2)$ where $d_3 = 2 = d_2(= 2)$ && $d_3 < d_1(= 5)$;
   - $(\alpha(4)$ where $d_3 = 4 > d_2(= 2)$ && $d_3 < d_1(= 5)$;
   - $(\alpha(5)$ where $d_3 = 5 > d_2(= 2)$ && $d_3 = d_1(= 5)$;
   - $(\alpha(6)$ where $d_3 = 6 > d_2(= 2)$ && $d_3 > d_1(= 5)$.
2. All new words composed by the original prefix and one of the suffixes are divided depending if they belong or not to the regular language studied.
3. Suffixes are organized in the form of a tree, where each branch is characterized by a complete transition function focused on the value of the corresponding suffix. Data symbol, guards, and registers are defined unequivocally. Isomorphic trees are merged and minimal SDT is therefore created.

Definition 2 for the Nerode equivalence cannot be used anymore since it does not contemplate the use of registers, trees, and data symbols. A new version of the Nerode equivalence needs to be described, generalizing the old one with the usage of SDTs and symbolic suffixes. The following is only a simplified version of the one proposed in [36] that manages to fully convey the operation of the Nerode equivalence.

Definition 13 ***(Symbolic Nerode equivalence)*** *Let* `O` *be a canonical tree oracle for a theory* $\langle D,R \rangle$*. Let* `u` *and* `w` *be data words. For a language* $\mathcal{L}$*, the words* `u` *and* `w` *are Nerode-equivalent if for any set* `V` *of symbolic suffixes, there is a bijection function* γ *between the registers in* $O_{\mathcal{L}}$ `(u, V)` *and* $O_{\mathcal{L}}$ `(w, V)` *such that* $O_{\mathcal{L}}$ `(u, V)` $\equiv_{\gamma}$ $O_{\mathcal{L}}$ `(w, V)` *[34].*

Two data words are equivalent with respect to a data language $\mathcal{L}$ if and only if their SDTs are isomorphic.

Nerode equivalence works similarly with the original one presented in Section 2.2.1. Also in this case, the symbolic Nerode equivalence can be used to separate locations in the building phase of the RA. A symbolic version of the Myhill-Nerode theorem is used to build a register automaton where each state directly corresponds to (at least) one symbolic Nerode-equivalence class.

Definition 14 ***(Symbolic Myhill-Nerode)*** *Let* $\mathcal{L}$ *be a data language, and* $\equiv_{\mathcal{L}}$ *a symbolic Nerode equivalence on* $\mathcal{L}$*. If* $\mathcal{L}$ *is regular, then we can build a register automaton that recognises* $\mathcal{L}$*. This automaton will have at most as many locations as the number of equivalence classes of* $\equiv_{\mathcal{L}}$*.*

2.5 SL* learning algorithm

SL* algorithm is an implementation of the already represented L* learning technique. It has been implemented in [36] specifically to learn an RA for an unknown regular data language inferring locations, registers, and transitions.

It follows the canonical pattern for active automata learning. It uses the same learning structure as the L* algorithm, using the interaction between *teacher* and *learner*, queries and observation table. The *learner* originally does not have any knowledge of the case study and treats it as a black box.

Queries are developed similarly to those used for the L* algorithm. Two types of queries are answered by the *teacher*, one to understand the behaviour of the system and one to check the equivalence of the hypothesis with the real model. Those are called tree queries and equivalence queries.

Tree queries substitute what are membership queries for the L* algorithm. Tree queries permit to enlarge the knowledge of the system. The scope of those queries is to build the observation table that will lead to the creation of hypotheses of the model. The *learner* submits a concrete prefix and a set of symbolic suffixes to the *teacher*. Answers are proposed employing a tree oracle. Formally, as soon as the *teacher* receives a tree query, it forwards the query to the tree oracle. It transposes it to several membership queries to build the SDT that will represent the answer of the tree query [45]. The answer is saved inside the observation table that is continuously updated. The *learner* continues to make queries until the table is complete and a hypothesis could be created.

The observation table is again the central element used to keep track of the knowledge level of the learned system and it is needed to produce the hypothesis automaton. The concept that found it is the same of the one in the L* algorithm. It stores data words and the corresponding SDTs that extend the prefixes.

Definition 15 ***(Observation table for RA)*** *An observation table is a tuple* $\langle$ `U`, U^+, `V`, `Z`$\rangle$*, where:*

- `U` *is a prefix-closed set of data words, called short prefixes;*
- U^+ *is a set of extended prefixes, each of the form* $u\alpha(d)$ *for some* `u` $\in$ `U` *(and, in general, not disjoint from* `U`*);*
- `V` *is a set of symbolic suffixes; and*
- `Z` *is a function that map each element in* `U` *and* U^+ *(*`U` $\bigcup U^+$*) to a SDT in the form of a* `(u, V)-tree` *[36].*

The RA observation table must also take into account the register trends. Registers have to be saved consistently with the model, using the following definition.

Definition 16 ***(Register-consistent observation table)*** *An observation table is register consistent if for each extended prefix* $u\alpha(d) \in U^+$*, whenever* d_i *is a data value in* `u` *and* $x_i \in \chi(Z(u\alpha(d)))$*, then there is also a register* $x_i \in \chi(Z(u))$.

The complete table is divided horizontally into two parts, with short and the extended prefixes. The boundary is shown as a double line. Like in the case of the L* algorithm, the difference between short and extended prefixes is related to the content of the corresponding cells of the table.

Every new studied prefix is supposed to be extended with symbolic suffixes. If one of them has a behaviour that is not represented by one of the short prefixes, it is promoted to the upper side of the table. Since it should save the SDT provided from the tree oracle as the results of the tree queries, the structure of the table is composed only from 2 columns. One contains the prefix and the other one contains the corresponding symbolic suffixes.

The *learner* fills the symbolic suffix cells of the observation table `Z(u)` with the `(u,V)-tree` generated by the oracle. Every time the SDT should be changed, it is updated to the new version and the set of symbolic suffixes is updated. The first prefix to be studied is ϵ and every important suffix is controlled through queries until the table is closed. If new SDT are found, they are promoted to the short prefix section. New cases are then studied to close the table. An observation table is said to be closed:

Definition 17 ***(Closed observation table)*** *An observation table* `O` *is said to be closed if for any prefix* `s'` $\in (S \bigcup (S \cdot \Sigma))$*, already exists a short prefix* `s` $\in$ `S` *such that* `s` $\equiv_O$ `s'`.

As soon as the observation table is close and register-consistent, it can be used to build an hypothesis of the studied system.

Definition 18 ***(Hypothesis automaton)*** *A closed and register-consistent observation table* $\langle$`U,` U^+`,` `V,` `Z`$\rangle$ *can be used to construct a hypothesis register automaton* `Hyp(`$\langle$`U,` U^+`,` `V,` `Z`$\rangle$`)` $=$ $\langle Q, q_0, F, \chi, \Gamma\rangle$*, where:*

- *the set* Q *of location is obtained from the set* `U` *of short prefixes: each short prefix* `u` $\in$ `U` *represents a node* $q_u \in Q$*;*
- *the initial location* q_0 *is represented by* ϵ*;*
- *the set of the final accepting locations is* F*. It is composed by all state* q_u *accepted accordingly with the state* `Z(u)`*;*
- *registers* χ *maps each location* `u` $\in$ `U` *to* χ`(Z(u))` *(note that* $\chi(q_0)$ *is the empty set);*
- *each* $\alpha - transition$ $\langle q, \alpha(d), g, \pi, q'\rangle$ *of* `Z(u)` *generates a consistent transition* $\langle$`u,` $\alpha(d), g, \pi',$`u'`$\rangle$ *in* Γ *where:*
  - `u'` *is the unique short prefix in* `U` *such that* $u\alpha(d_u^g) \equiv_\gamma$ `u'`*;*
  - $\alpha(d)$ *is a parametrized input symbol with data value* `d`*;*
  - g *is a guard over* `d` *and* $\chi(d)$*; and*
  - π' *is an assignment*χ`(Z(u'))` $\rightarrow$ $($$\chi$`(Z(u))`$\bigcup$ `{d})`*. For* $x_i \in \chi$`(Z(u'))`*, we define* $\pi'(x_i) = \gamma^{-1}(x_i)$ *if* $\gamma^{-1}(x_i)$ *stores a data value of* `u` *in* `Z(`$u\alpha(d_u^g)$`)`*, and* $\pi'(x_i) = d$ *otherwise [30].*

This definition is used to construct a hypothesis automaton starting from a closed and register-consistent table. If the table is not closed, it means that an extended prefix presents a behaviour SDT, that is not identified by any other short prefix. Therefore, such prefixes simply need to be promoted to the upper part of the table (short prefixes). If the table is not register consistent, a SDT built for $\mathtt{u}\alpha(d)$ has a register x_i that expects a value from $\mathtt{u}$, but the SDT $\mathtt{Z(u)}$ does not have the register to store the specific value. Extending the set of symbolic suffixes and updating the $\mathtt{Z}(\mathtt{u}\alpha(d))$ solve this problem. The specific suffix that uses the missing register x_i must be added to $\mathtt{V}$ as α_v.

The created hypothesis automaton is submitted through equivalence queries to the *teacher*. Equivalence queries are used to control the consistency of the model hypothesis against the real model. The *learner* submits the hypothetical version of the automaton created with the knowledge summarized in the observation table. The control is on the adherence of the proposed model to the studied system. Theoretically, the *teacher* answers with a positive statement if the model is correct, otherwise it provides a counterexample.

Practically, since in a black-box scenario, the *teacher* does not have a complete knowledge about the component's behaviour, so equivalence queries cannot be directly addressed. New methodologies need to be used to propose the correct answer to those types of queries. Usually equivalence queries are approximated by some form of search for a counterexample.

Search strategies can help in the solution to this problem, some examples are provided below.

- **Exaustive exploration** is a technique that searches in the set of data words for a fixed and limited amount of time for practical divergence. It is a costly strategy but presents two features that are strategical for a proper operation of the learning algorithm. It guarantees that only the shortest counterexamples are found and that, at every time the current hypothesis is definitely correct for words up to some specific length.

- **Random walk** presents less guarantee on the performance of the process, but have proven to be a simple yet very effective strategy for finding counterexamples fast [21, 46]. There is no guarantee that the provided counterexample is the shortest one. In the initial phase, it is not a problem, but especially for complex systems, it can lead to long symbolic suffixes that raise the computational effort required to terminate the learning process. Both [21, 46] prove that removing loops from counterexamples have a very effective heuristic for controlling the effect of long counterexamples on the performance of the learning algorithm [36]. For each studied hypothesis the random walk algorithm was started three times and the shortest counterexample was used. Other bounds have been imposed on the random walk algorithm, e.g., the maximum number of walks or tests performed for each experiment. In this way it is plausible that the result is short, although no guarantee to be the minimal version can be provided [21].

In any way the counterexample is provided, it should be analyzed to solve the specific case. It could represent a missing transition over an existing tree, or a missing location `U`, or an incorrect allocation of γ between different SDTs while constructing the hypothesis. All those cases can be solved by adding the specific suffix of the counterexample to the set `V` of symbolic suffixes of the observation table. Finding which suffix to add is another challenge.

The counterexample should be divided into a short prefix and the suffix to be added to `V`. The break point of the work is in the place where the old SDT diverges from the one proposed as counterexample. As soon as the suffix is found, it is added to the set of symbolic suffixes, queries are computed again, and the observation table updated. This process is repeated cyclically until no counterexamples are found and a minimal, in terms of locations and registers, RA for the language $\mathcal{L}$ is proposed. Note that the minimal number of locations and transitions is not an absolute value but varies depending on the particular theory and tree oracle used.

Presenting how some iterations of the learning process are done could be propaedeutic to properly understand how RA are learned. The FIFO-constrained example has been chosen to maintain the consistency in the presentation. For clarity, we deal here only with one of the two possible functions (`?push[int]`). The same reasoning can and should be extended to the other function (`?pop[ ]`). The chosen theory for this example is equality over natural numbers extended with constants.

The *learner* starts sending the first tree query for the prefix ϵ and the set of symbolic suffixes $\{\epsilon\}$. The resulting tree is added to the observation table. The table should be closed and consistent to permit the creation of a model. A new tree query is performed for the ϵ prefix concatenated with the symbol $push(d)$. A row is added in the extended prefix part. The SDT corresponding to this query is isomorphic to the other one, so the prefix remains in the extended prefix part of the table. The table is therefore closed and register consistent.

The *learner* can easily create a hypothesis automaton following the same process presented in Definition 18. An equivalence query is provided and checked during the hypothesis validation phase. The model does not fit the system, so the *teacher* provides a counterexample where the two are not

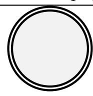
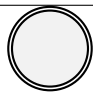
Observation table	
Prefix	Symbolic Suffixes
	$V = \{\epsilon\}$
ϵ	
push[d]	

Table 2.2: SL^* first observation table on FIFO system.

consistent. The counterexample could be, for example, `push[c1 == d1]` that is not in the target language, but accepted by the hypothesis. This case represents a missing transition over an existing tree. The push function is provided in the table, but no differentiation on data values is provided.

Processing the counterexample, a new symbolic suffix is retrieved and added to the set V. The table has to be updated consistently studying all new cases and correcting the existing SDT to obtain again a close and consistent table.

This table is close and consistent. It is only been presented the system triggered from the push function, leaving aside the pop function that would have raised the complexity and made difficult the comprehension of the observation table and counterexample processing phases. Looking at the whole system after two passages it is possible to learn the complete model of the system.

Here below it is proposed the final hypothesis characterizing the FIFO system with the constraint regarding the push function with the data value equal to 2 and a one-dimension array (only one available element in the system).

This model represents an automaton created similarly to a Mealy machine because the output function is labelled directly on the transition. The transition between two consecutive states is divided in two parts to highlight both the transitions that trigger the change in state and the output state corresponding to that transition. The intermediate state is fictitious and needed to merge those transition that are characterized by the same output function. It is therefore used as a graphical loophole to simplify the representation. Transition representations are simplified in the graphical exposition and comprehend a merged statement between the data word and the guards.

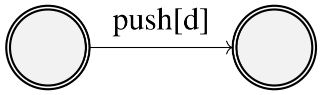
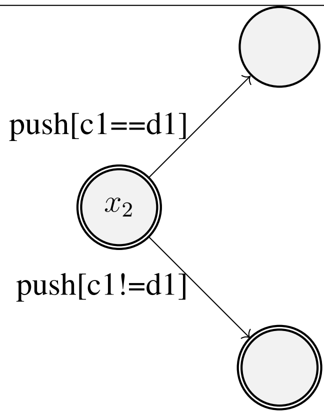
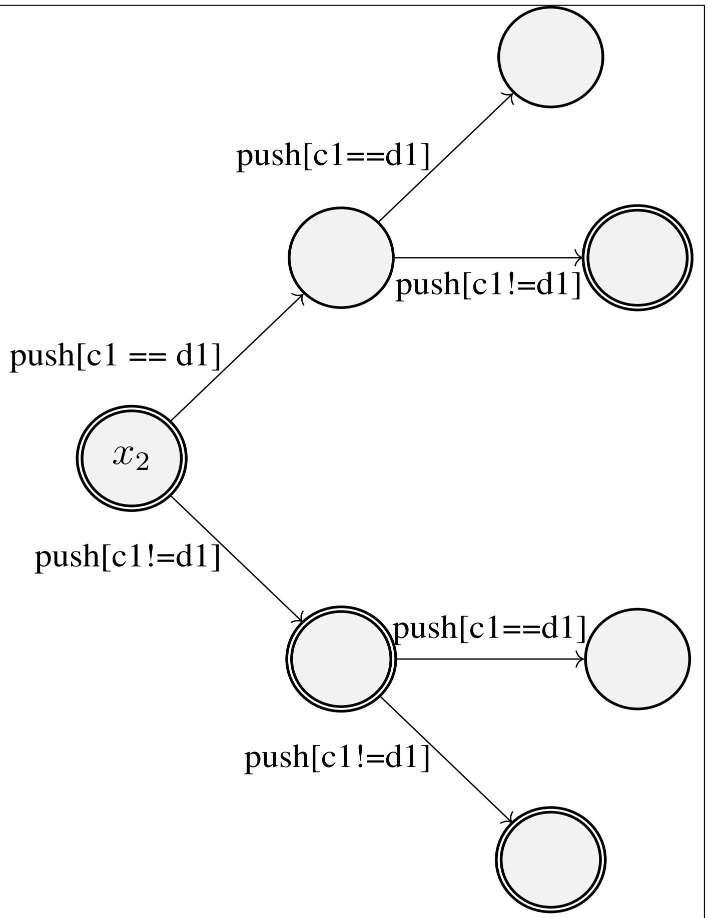
Observation table	
Prefix	Symbolic Suffixes
	$V = \{\epsilon, push[d]\}$
ϵ	
push[d]	
push[d] push[d]	

Table 2.3: SL^* second observation table on FIFO system.

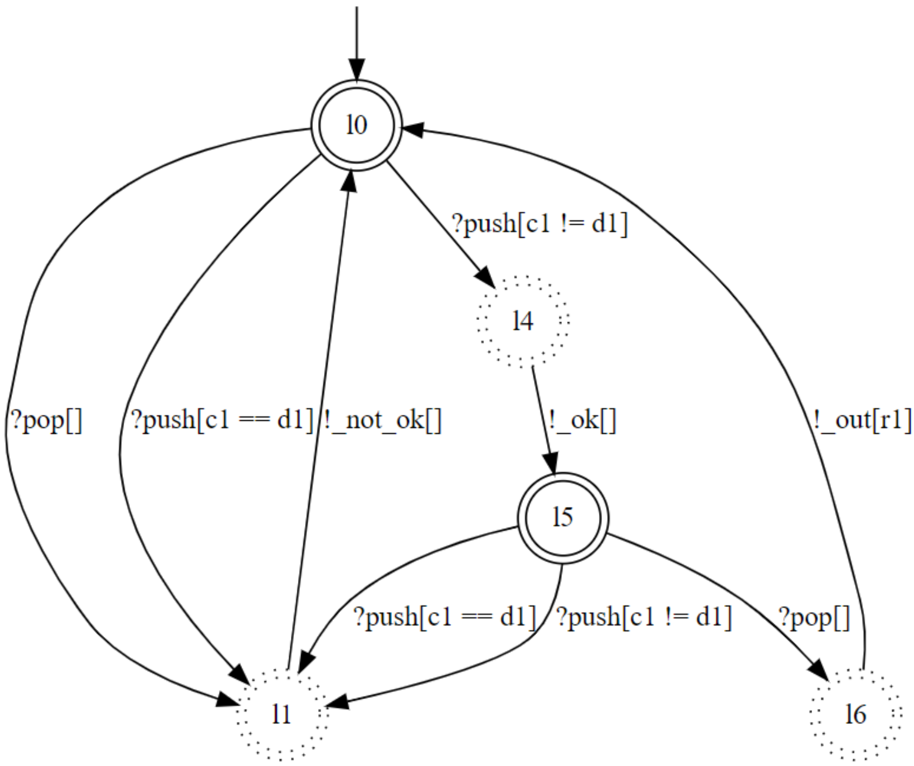


Figure 2.10: Hypothetical model of a FIFO system with constraint.

Nodes are defined as the states of the FIFO system characterized by the number of occupied elements in the array. There will only be two nodes that respectively defines the empty system and the system with one element (full). The initial accepted state is called `l0`. It characterizes the empty system. The function `?push[c1 != d1]` is the only one that bring to an accepted state (`l5`). It passes through the intermediate location `l4`. It is the function where a push request is made with whatever integer data value different from the forbidden value `2 = d1`. The other inputs to the systems are not accepted. `?pop[ ]` function can not be performed because the system is still empty. `?push[2]` is forbidden by construction. Both these functions are not accepted and therefore bring back to the original state `l0` consequently an error statement. Moving from `l5` every `push` function that respects the constraint and does not exceed the maximum dimension of the system creates a closed loop with accepted state on `l5`. If the `push` function does not respect the constraints it enters in the `!_not_ok[ ]` cycle. Specifically our system is full after one iteration so every other `push` function is not acceptable. `pop` function starting from `l5` brings back the accepted state to `l0` and outputs the value saved in the register `r1`.

2.6 Learnlib & RALib

Many recently developed libraries permit to infer automata learning algorithms. One of the most used is *LearnLib* [47, 48, 41]. It is an open-source framework implemented in Java. It was developed mainly from researchers at *Technische Universität Dortmund*. It proposes a different methodology and algorithms based on a deep literature study, the actual state of the art and active research study from the developer. *LearnLib* provides various and always updated algorithms. Both passive and active algorithm are presented, as many approximation strategies. It is structured in blocks that can be interrelated and exchanged bringing to the widest range of experiment to be developed. We are mainly interested in active algorithms such as L*. Many others are developed such as ADT, DHC, DT, NL* or TTT* are proposed [22]. RALib [21] is developed as extension of the framework proposed in LearnLib [47]. It is mainly thought of as a tool for active learning of Extended finite automata (EFA) [49, 50] including RA. The SL* algorithm is the main focus of this library. It is an improved version of L* based on the main features that characterize RA. It creates an automaton with the interaction between the *teacher* and *learner* that poses queries and creates an observation table. Many changes have been made from the developer to fit the structure of an RA, like the definition of registers. The execution process of the algorithm, recalls the L* one, and is divided into three phases: Hypothesis construction, hypothesis validation, and counterexample processing.

* active automata learning algorithm formulated in the Minimally Adequate *teacher* framework

Chapter 3

The Lateral State Manager as a SUL

As mentioned in Chapter 1, the scope of this thesis is to learn an RA model from an AD system implemented in MATLAB. Therefore, an introduction to the SUL, called the Lateral state manager (LSM) is presented here. The LSM is also the focus of study in [1], which includes preliminary results that act as the starting point of this thesis and therefore makes it possible to compare the results from different learning algorithms for future work.

The LSM is part of a Decision making and Planning module in an AD vehicle and it is responsible for managing the modes during an autonomous lane change manoeuvre. The general structure of the AD system is based on separated blocks and Figure 3.1 shows a schematic representation of the same. All necessary information that is involved in the decision making and planning process are sent as input to the system. The `decision making and planning` system is here represented as a box containing sub-parts responsible for more specific tasks. The structure is composed from a `Planner`, the `LSM` and a `Path Planner`. Although the methodology developed during this thesis focuses only on a single sub-system, the LSM, it can, in the same way, be applied to every other block.

By way of example, the Decision making and Planning module has to ensure that a request for changing the lane will provide safe decisions at all times. Inputs of the decision process are the vehicle state and the surrounding traffic state. The `High Level Strategic Planner`, called for its simplicity `Planner`, is responsible for strategic decisions depending on the state of the surrounding traffic situation and the state of the AD vehicle. These decisions are communicated in the form of `lane change request` to the

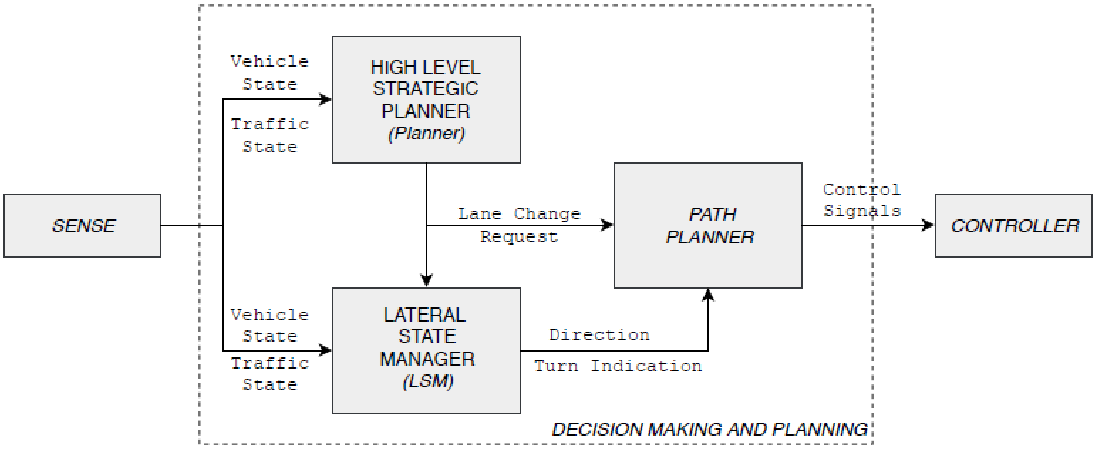


Figure 3.1: Decision Making and planning system scheme [1].

`LSM` and the `Path Planner`. The `lane change request` indicates the desired lane for the vehicle to move into and can take one of three possible requests: `NoRequest`, `ChangeLeft` and `ChangeRight`. The LSM considers the lane change requests together with the vehicle and traffic states and manages the different modes during the lane change process. Lastly, the Path Planner considers the outputs of the Planner and the LSM to provide the low-level control to the vehicle to achieve the desired trajectory.

The LSM is implemented in MATLAB as a finite state machine due to the characteristic structure of the task it solves. The implementation of the LSM consists of several variables and various function calls. In each execution cycle, the LSM undergoes three stages, where the inputs are first updated, then depending on the current mode a decision is made on whether the system transits to a new mode or not in the second stage, and in the final stage if a new mode is reached, the code assigns new output values to the variables. The internal mode that defines the state of the LSM consists of five variables. They are:

- `state` that defines the current state of the vehicle, and could take values such as state A, state B, etc. During the autonomous manoeuvre, each of these states represent a discrete mode during the lane change process;
- `direction` that takes values similar to the `lane change request` from the Planner described earlier to indicate the desired lane to change into;

- three boolean variables to keep track of status flags during the lane change process.

The above variables are all read and written during each execution of the LSM.

As described in [1], the LSM also consists of 11 other parameters that are taken into consideration for the decision process. They define the environment surrounding the vehicle or the vehicle itself. In total, there are 10 boolean decision variables and one enumerated variable with three possible values similar to `direction` and `lane change request`. The input alphabet Σ of the SUL is composed from all the combinations of the decision variables and the lane change request. Hence, the input alphabet has $2^{10} * 3^{1} = 3072$ elements. The value of all the decision variables can be joined in one single ordered vector that can later be decoded.

Such a big number of possible combinations of decision variables can cause an explosion of the complexity of the system. This could indeed be a problem for learning a model of the SUL and solutions need to be found. It can be solved in different ways in the learning process.

In Section 2.4, all the learning tools that can be used in the learning process have been presented. To ensure the convergence of the algorithm, all useful tools have to be used. It should be noted that the definition of constant parameters, different data-types, and specific theories for each data experiment is crucial to the learning process. Looking at the SUL code, it has been possible to understand which are the constants that trigger each variable in the system. All variables that present the same behaviour in the data-type domain can be merged under the same category.

Other types of assumptions can be made. They try to deal with the reduction in size of the alphabet, typically removing variables and limiting the number of possible combinations. Obviously, the variables need to be studied in detail, and patterns or recurrences need to be found to reduce the number of studied cases without compromising the effective operation of the system. We hope not to exploit the last type of assumptions because they are error prone in the same way as the development of the code, and therefore will limit the utility of the proposed technique. If wrong assumptions are made, it could happen that some cases are not studied and the system is not the desired one. We should, at the same time, be conscious that it is possible to not manage to train the full system, given the considerable dimension of the alphabet. A more detailed explanation will be provided with a practical example in Section 6.4.

Chapter 4

Methodology

In this chapter, methods used to accomplish the goal of this project are discussed. All steps are presented and ordered with the goal of permitting to reproduce the work analytically. The method is created along the lines on the one presented in article [51]. The chosen methodology was chosen to mainly define a verifiable approach to answer the research questions.

Starting from the definition provided by [51], we can state that this project followed different types of strategy depending on the stage of the development process. In a preliminary phase, an exploratory research strategy has been used. It consists in creating the best solution, exploring the different possibilities in order to maximize the quality of the results. Later on, inductive reasoning has been introduced. It aims to formulate a clear proposition starting from an initial observation as results of the exploratory research [51].

In the previous chapters, we understood the importance of finding the formal model of the SUL automatically to verify the control system of autonomous cars. The possibility of using RA has been highlighted, in Section 2.4 and 2.5, as a promising solution of this problem. Many different libraries and tools can be used to construct a formal model. Various libraries have been developed during the years as presented in Section 2.6. We previously presented some general reasons on why *RALib* can be a possible solution for our case study.

The task has been divided in smaller steps to avoid the possibility of errors and to provide the reader with a clear understanding of the development process. The first point was an investigation on how to create automata learning models. A focus has been given on RA for the reasons explained above (Section 2.5). Some illustrative examples have been created to show how to use RALib as a learning framework. An important point was the creation and establishment of an interface between the learning algorithm and a general studied system. Finally a suitable learning algorithm to learn a RA model of the SUL has been implemented. The learning algorithm has been used on some benchmark examples and finally the LSM code. Every studied example needs some adaptations to obtain the desired solution. Some assumptions have been made to improve the results and obtain the aimed solution. Finally, the learnt model has been validated and the reasons for any shortcoming have been investigated.

The expected scientific result, that this thesis aims for, is an automatically generated formal model of the SUL. Given the literature study, in Chapter 1, 2 and specifically in Section 2.5, the ideal result could be the convergence of the full algorithm to a model. In the case in which the methods proposed would not have been sufficient to accomplish the goal and some general errors have been retrieved, increasing abstractions have to be introduced. This will still be a satisfactory output.

The final algorithm has been developed in such a way to ensure the correctness of the final model. Data and models have been collected and analyzed with a qualitative method. In the validation stage, the qualitative features, above all the **finishing the training process**, have to be translated into quantitative data for having a useful research with measurable outcomes. Once the validation of the model is done, it will be possible to check the correctness and safety of the code and therefore of the proposed behavior of the vehicle.

In the event of a failure to learn an appropriate model, investigation of causes for such a failure and insights on potential solutions are also considered part of the evaluation. Insights and results from this project can then be used to scale this approach further. This could potentially lead to the formal verification of a complete system.

4.1 Research tools

The project was developed in the form of software. A hardware tool, however, has been necessary. The computer used to develop and train the algorithm is a *HP ELITEBOOK x360 830* late 2020 with 10th Generation Intel®Core™ i7 processor, 16 GB memory, 256 GB SSD storage unity capacity. It was equipped with *Linux Ubunutu 20.04*.

From the software point of view, different libraries and programs have been used. The LSM algorithm has been developed in *MATLAB* . The R2021a MATLAB version was used. The learning algorithm has been developed using *Eclipse* as an integrated development environment. It is largely defined as one of the two best platforms to program with Java. The used version of *Eclipse* was 2019-12. The Java version employed was:

```
openjdk version "1.8.0-292"
OpenJDK Runtine Environment (build 1.8.0-292-8u292-b10-0ubuntu1~20.04-b10)
OpenJDK 64-Bit Server VM (build 25.292-b10, mixed mode)
```

The two studied libraries are *LearnLib* and *RALib*. Both of them have been updated and maintained *as is* at the date of the 10/05/2021. Until today, both libraries did not receive further updates in the used files. Some updates have been proposed for files in the library that are not related to the proposed work.

4.2 Experiment and data analysis

Many experiments have been designed to split the project task into smaller steps. The aim of all of them is to create a unique methodology that will permit to implement and verify the final objective of the thesis. Different learning examples have been tested in order to find a setup that could converge to the final solution proposing the desired output.

The development phase was divided into four steps. Every step contributes a fundamental component of the final version of the algorithm. This method was implemented so that every implemented phase adds something new and provides a sort of insurance on the previous steps. The four experiments were studied to specifically learn and test the following topics:

1. MATLAB-Java interface;
2. test of the library thanks to a benchmark canonical FIFO example;
3. usage of integer equality theory with constants over a FIFO system modified to impose constraints;
4. Learning the LSM.

The first experiment consists in the development of code in Java that manages to communicate (I/O communication) with MATLAB. Several tests with growing complexity have been carried out to propose a safe and unique solution.

To test the library operation, the FIFO benchmark example has been carried out. The proposed solutions were developed and presented from the simplest to the most complex and complete example. The first example has been developed in Java language to check only the RALib working process. Then the same example has been developed in MATLAB introducing the MATLAB-Java interface developed above. This made possible testing simultaneously the validity of the library and of its communication.

In the following experiment, the FIFO with constraints as presented in section 2.5, has been used as SUL of the learning algorithm. Constants have been introduced in the learning theory. This example is therefore closer to the LSM that uses constant parameters in the same way.

The last experiment is to extend the interface so that a model of the LSM as SUL can be learned.

4.2.1 System validation and verification

The intrinsic structure of automata learning algorithms provides itself a validation and verification of the system. The usage of queries, counterexamples, and hypothesis validation ensure the correctness of the proposed system. Each model created from an automata learning algorithm is not incorrect. Instead it is possible to decide that it does not satisfy our need because it does not capture as much behaviour as we want.

Assumptions can be made to achieve a solution. Those can bring a loss in generality since we are not studying all cases. Using a simple example: if we decide to not study one of the variables in the system, all states that were created thanks to a change in that variable are not studied and therefore not modelled. The final model represents the studied system without some features. Assumptions should be avoided because they can manipulate the logic of the system, but sometimes are needed to ensure the convergence of the algorithm to a solution.

Manual and visual comparison of the solution and of the graphs provide a double check of the validity of the solution. Some data relative to the learning process are provided and can help in understanding the learning algorithm. Also the development process, the learning setup, and the methodology described here helps in providing examples of the correctness of the solution.

Chapter 5

Learning setup

To answer the research questions proposed in the Section 1.3, we suggest a learning setup that describes how the SL* algorithm is supposed to interact with the SUL and how this environment will create a solution to the addressed problems [52].

The description is divided into two steps. To begin with, a high level learning setup, then a more detailed explanation of the code is presented. All software design choices to build the full SUL will be presented in detail.

5.1 High level learning setup

The learning setup is structured into three parts that together compose the working algorithm. The formal model is the final output of the full learning setup and can be seen as the last block of Figure 5.1.

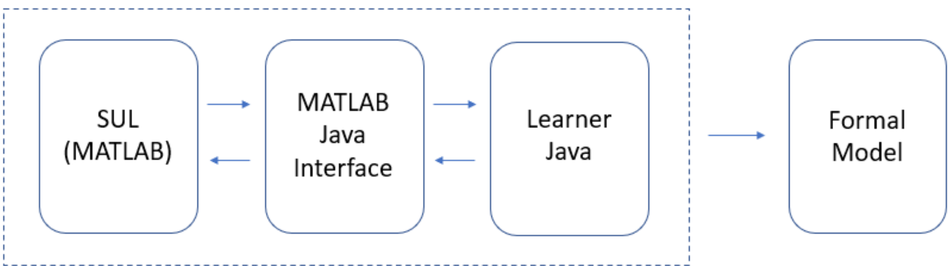


Figure 5.1: Overview of the learning setup [1].

The learning setup is divided into three blocks: the SUL, the learning algorithm, and the interface that connects them. The first element defines the

SUL, in our case coded in MATLAB. The learning algorithm is instead based on the RALib library in Java. To connect these two parts, an interface has to be created.

To actively learn an RA model of the SUL, it is intuitive, from what was explained before, that a SUL and a learner are needed. In Figure 5.1 the learner is the part supposed to create the model, implementing the SL* algorithm as presented in Section 2.5. The SUL represents the part of the learning setup with the knowledge of the behaviour of the system.

The learner has been introduced in Chapter 2 and Section 2.5. The SUL has been presented in Chapter 3. The central part of Figure 5.1, the so called *MATLAB-JAVA interface*, is the one that has still not been presented. Its scope is to connect and create a stable communication between the other two parts of the setup. This part is needed because the learner and the SUL have been developed in two different programming languages. The SUL has been developed in MATLAB. The learner is implemented in Java.

An interface is therefore necessary to enable the communication. The queries are proposed from the learner side by the oracle and should be transformed in the language of the SUL to obtain an answer. This communication process enables the execution of specific requests and commands to and from the SUL. The state and the outputs produced from the system should be observable. We will further discuss the implementation of the MATLAB-Java interface.

It is important now to highlight that error messages need to be proposed if the required action is not executable by the system. As said before, this code is intended to solve a more general problem than the one specifically presented for the LSM. This implementation can be therefore be used also for any other SUL. Some adaptations will be needed depending on the language of the SUL and the specific code. The following subsections will present the software design.

5.2 Software design

In Chapter 4 the four experiments used to answer the research questions are described. These have been divided into progressively advanced steps to implement a safe and clear methodology. This section presents the software design and the developed files to implement the most important sections of the learning setup. The two following subsections explain the MATLAB-Java interface and the learner (Section 5.2.1). The learning algorithm is created thanks to different files in the RALib library. The most important files used to develop the SL* algorithm are presented in Section 5.2.2.

5.2.1 MATLAB Interface

The MATLAB-Java interface is needed to define the interaction between a generic MATLAB SUL and a Java learning algorithm. As explained in Chapter 3, the SUL has been developed in MATLAB in the form of an automaton.

MATLAB proposes the possibility to integrate every native code with some external programming language and systems both in synchronous and asynchronous ways. Different languages and versions are supported. Java is one of the languages supported. Specifically, Java 8 (Oracle Java) is said to fully support the desired connection. Mathworks website proposes full and precise documentation on how to set this communication [53, 54, 55, 56, 57].

MATLAB can be *called* from another language using an Application programming interface (API) called in the specific case: Engine Application. This API permits us to call MATLAB functions from our application. Engine API for Java is provided with the standard user version and usually located in `MATLABroot\MATLAB\R2019b\extern\engines\Java\jar` [56]. To implement the interface in Java code with MATLAB, the following steps are needed:

- **Import the desired package**:
  import com.mathworks.engine.;*
  import com.mathworks.engine.EngineException;
  import com.mathworks.engine.MATLABEngine;
  import com.mathworks.types.Struct;

- **Start communication between Java and MATLAB thanks to the command startMATLAB**:
  MATLABEngine eng = MatalbEngine.startMATLAB();

- **Evaluate and receive functions and their outputs with the argument of the function feval**:
  st = eng.feval("LSM.m");

- **Close the communication**:
  eng.close();

More detailed explanations can be found in the MATLAB website [53, 54, 55, 56, 57]. Checking the correctness of the code and avoiding as much as possible errors and wrong results is a fundamental part of the programming phase. It is also crucial since the proposed application is safety critical. Errors or unexpected events that occur during the program execution are called exception. Those may be caused both by software or hardware errors. A good code should include exception handling mechanisms to anticipate errors. Many programming languages, like ADA, C++, and Java, recognize that exception handling is now a normal phase in the development process. Java presents some built-in tools developed to deal with this type of errors. Other exceptions, different from the provided one, can be controlled with a few lines of code. Signalling exception is achieved with a **try** statement. The handler part is instead handled with the keyword **catch**.

The most difficult errors to be caught are communication errors. These types of errors are caused by wrong input requests or other unforeseeable problems. Many other errors should be taken into account. In this section, we mainly focus on the errors that can occur during the communication between Java and MATLAB. Usually, when an error occurs in Java, error messages are automatically generated. In the case of external communication, like the one presented above, further checks need to be done. It is not possible to directly throw the built-in exceptions, because the native classes where the communication is based, do not implement exceptions. A *try-catch* structure has therefore been implemented in order to check *InterruptedException* and *ExecutionException*. According to [58], *InterruptedException* is: "Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity". "Execution exception is thrown when it is attempting to retrieve the result of a task that aborted by throwing an exception" [59]. No other exceptions are shown to create problems in the evaluation of the code, so are not implemented. The *try and catch* structure has been proposed in the following way:

Listing 5.1: Try and catch structure in Java.

```
try{
    //code to be verified
} catch (InterruptedException e) {
  System.out.println("Interrupted Exception");
  System.out.println(e.getMessage());
} catch (ExecutionException e) {
    System.out.println("Execution Exception");
    System.out.println(e.getMessage());
}
```

5.2.2 Learning algorithm

The right part of the learning setup, as presented in Figure 5.2, represents the learner. It implements the SL* algorithm with all necessary details and features that may be used.

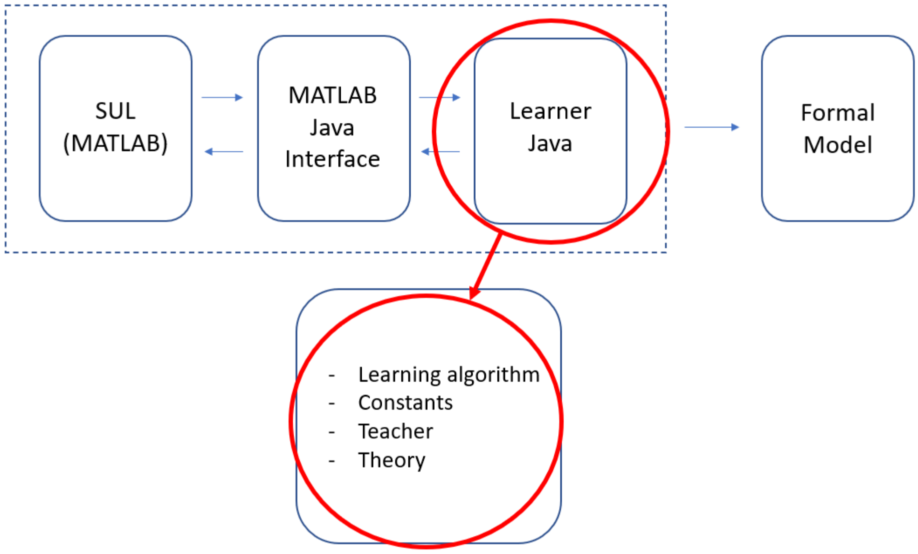


Figure 5.2: Focus on the learner's components in the learning setup.

The learning schedule is clearly defined and implemented in the function `runExperiment`. This function can directly launch the course of action. The algorithm is built following what was presented in Chapter 2 and Section 2.5. It constantly iterates between the hypothesis construction, hypothesis validation, and counterexample processing phases until counterexamples are no longer found Section 2.5. As shown in Figure 5.2, the learner contains the definition of the work flow of the SL* algorithm.

To start the learning process, it is needed to fully characterize the learning environment and the system. By way of example, some required inputs are the `SUL`, the `teacher`, the `constant` and `theory`. All these variables and others are defined as input parameters of the `main` function. The SUL is directly retrieved from the study case through MATLAB, thanks to the function: `matEng.feval("LSM")`. Other variables are directly created from existing files in the library that have not been manipulated.

We believe it is important to describe some functions of the algorithm. The teacher is one of the variables that mostly influences the operation of the algorithm. The teacher is defined thanks to a complex structure. It should connect a specific theory to the corresponding data type studied. It is based on a linked hash map structure to create a map `Map<DataType, Theory>`. More data types and theories can be defined inside the same system if it is needed to better represent different variables.The chosen theory has to be consistent with the data type used and the guards that the SUL implicitly defines. Data types could be any class of numbers (e.g., integers or real numbers). Only the data types developed in the library can be used. Specifically RALib does not support boolean numbers as data type. Many RA in the literature only allow equality theory imposing restrictions on the possible modelled components [60, 61]. In [30], extensions for RA formalism are proposed and new theories presented and consequently implemented in RALib. Both equality and inequality theories can be considered.

As the case study is implemented, the equality theory over integers best fits the SUL. Inequality theory would not have been a feasible solution since the SUL is not sensitive to inequality checks over the data values that characterize the transitions. For this case study, we only used integer data types and the equality theory over integers. When needed, constant parameters have been introduced to simplify the learning process and reduce the number of studied cases. For example, boolean variables that have to be modelled with integer numbers need the definition of a constant to avoid useless computations.

The guard is therefore implemented only with equality and the data values only belong to the set of integer numbers. Each transition is represented by the data value that characterizes only the specific passage and the constant parameter Section 2.4.

The hypothesis construction follows the definitions in Section 2.4 and 2.5. Once the observation table is close and consistent, and a hypothesis is created, it needs to be verified. The equivalence check has been implemented using the *random walk* method because, although it does not provide the guarantee of a minimal counterexample, it is much faster and permits a bigger system to be learned. On the other hand, the exhaustive exploration does not manage to find the shortest counterexample in an acceptable and comparable predetermined amount of time. In papers such as [46, 21] a random walk method has proved to be successful and efficient, especially when investigating counterexamples for loops before submitting the counterexamples to the learning algorithm.

An important point, introduced in a second time in the library, is the process that permits to convert the hypothesis created at every iteration of the algorithm to the corresponding model. It has been possible thanks to the

`RAToDot` file of the library. The function saves a dot file and permits the visualization of the model thanks to the graphviz tool. Dot files delineate the graph of the produced register automata at every iteration. They define in a structured way every node and the transitions that connect every state. They define a hierarchical directed graph. Graphviz is an open source tool. It is a graph visualization software. A default command that permits the creation of the graphical representation of the model is:

```
dot -Tps model.dot -o model.ps
```

Chapter 6

Learning examples

In previous discussions, in Chapter 2 and 4, the construction and the working of the learning algorithms have been described. To understand and demonstrate the operation of the algorithm, several experiments with growing complexity have been performed. A detailed explanation of the development workflow is given in Chapter 4. Four experiments have been proposed. The scope of the first experiment, in Section 6.1, is the development of the MATLAB-Java interface. The communication has been tested with multiple simple test cases. For the second experiment, Section 6.2, the canonical FIFO example has been presented to understand the operation of the learning algorithm and to validate the MATLAB-Java communication. In Section 6.3, the FIFO example has been studied imposing the constraints presented in section 2.4 to permit the testing of constant parameters in the learning algorithm. In the last section, Section 6.4, the SUL example has been developed using all the information gathered from the three past experiments.

For each learning experiment, the most important data of the experiment are presented in a table. The presented parameters can be explanatory for the performance of the learning algorithm. Those represent both the performance of the learning algorithm and some parameters of the proposed model. The name is pretty self-explanatory. The studied parameters are:

- **Resets** represents the number of resets occurred in the learning phases;
- **Inputs** is the number of studied inputs;
- **Locations** is the number of locations in the final model;
- **Transitions** is the number of transitions existing in the final hypothesis;
- **Input Locations** is the number of locations in the system;

- **Input Transitions** represents the number of transitions that exit from all locations;
- **Build time** represents the amount of time used by the learning algorithm to build the final hypothesis.

The most important parameter to be used to understand the performance of the algorithm is undoubtedly the **Build time**. It indirectly depends on the number of queries, the length of the counterexamples, and the number of resets occurred. The first check to be performed to compare two supposed equivalent models could be on the number of **Inputs**, **Locations**, **Transitions**, **Input Locations**, and **Input Transitions**. To have a better comparison, the effective models should be compared.

6.1 MATLAB-Java interface

The first tested example has been the communication between MATLAB and Java. A simple mathematical function has been implemented in MATLAB and tested. The communication presented before (Section 5.2.1) has been employed. The case studied was an application of the `sqrt` function.

Listing 6.1: MATLAB function to evaluate the square root of the desired x value.

```
function z = Sqrt(x)
    l = length(x);
    for i = 1:l
     z = sqrt(x);
    end
end
```

The function accepts an input `x` that can be both a number or an ordered sequence of numbers. With a simple `for` loop over the length of the input is possible to retrieve the square root of every desired value. Negative numbers are acceptable from the original function `sqrt` and the result is disposed as a complex number. Since positive integer numbers are the only data type we are interested in for future experiments, we previously imposed constraints on the sign of the input.

The communication has been tested using the Java pseudo code in Listing 6.2. It is supposed to communicate an order sequence of numbers (`double[] b = {1.0, 4.0, 16.0};`) and output the square root of the elements in `b`. The example starts the communication with MATLAB and afterwards sends the desired inputs to the function. The output is then retrieved and printed [57].

Listing 6.2: MATLAB-Java interface to solve the Sqrt function.

```
import com.mathworks.engine.MATLABEngine;
...
public class App {
    public static MATLABEngine matEng;
    public static void main(String [] args) throws
       Exception{

        matEng = MATLABEngine.startMATLAB();
        matEng.eval("cd '~/MATLAB_files_location'")

        double[] b = {1.0, 4.0, 16.0};
        double[] roots = matEng.feval("Sqrt",b);

        int lb = b.length;
        for (int i = 0; i<lb; i++) {
            System.out.println(roots[i]);
        }
        matEng.close();
    }
}
```

The numbers `b = {1.0, 4.0, 16.0}` have been tested and the solution is `roots = {1.0, 2.0, 4.0}`. Other experiments have been made and we can confidently say that the outputs correspond exactly to the ones expected and therefore we can assume that the communication works properly [57].

6.2 Learning canonical FIFO

As presented in Chapter 4, the next fundamental point to be understand is the functioning of the learning process. This experiment has firstly been implemented in Java to only check the operation of the learning algorithm on a canonical FIFO system. The same exact FIFO example has been implemented in MATLAB to control the operation of the algorithm together with the interface. The experiment is considered successful in the case in which both the learning process terminates and the two solutions coincide. If the solutions coincide, the learning algorithm has successfully been merged with the MATLAB-Java interface.

The used example is a simple FIFO system. We have already shown why a FIFO system can be seen as a benchmark example for register automata. Both the development in Java and MATLAB are structured in the same way to ensure comparison fairness. The system was divided in three separate functions:

- `initialization` of an empty vector with maximum fixed capacity;
- `push` function that can insert integers values progressively inside the defined vector;
- `pop` function that returns the oldest value in the vector.

The development is schematized in Table 6.1 for both languages. The two columns describe the development of each function both in Java and MATLAB languages.

The learning process is the same as the one described in Section 2.5. The two most important developed files are the `App.Java`, that defines the main function, and `FIFOSUL.Java` that describes the interface between the SUL and the learning algorithms. This file is only exploited for the MATLAB case. No constants on the data values are needed for this experiment, since any difference in data values does trigger a change in the final state.

The learning experiment has been initialized with a fixed capacity of three elements. It is not consistent with the original benchmark example proposed in Section 2.4 where only one position is available. We believe that having a bigger system size can provide the reader a better understanding of the learning algorithm and model. Size three is a trade off between the dimension of the model to be learned and the computational time required. Larger sizes takes greater amount of time and do not add any important value to the learning process.

Both the experiments managed to fully converge and produce a solution. The two evaluations result in the same exact output. As mentioned in Section 5.2.2, the algorithm proposes a dot file and a graphviz diagram that represent the learned model. The following two images represent the two models retrieved from Java, in Figure 6.1, and from MATLAB, Figure 6.2.

After a careful study of the output, both the graph and the hypothesis code formulation, we can assume that the outputs are equal and both respect the system described in the SUL code. Through these two experiments, regarding the FIFO benchmark experiment, we understood:

Java	MATLAB
this.fifo = new ArrayList<>(capacity);	theQueue = int32.empty(0,capacity);

public boolean push(Int value)
if (fifo.size() < capacity)
fifo.add(value);
return true;
else
return false;
end
end
	if lenght < capacity
	if isempty(var)
	sys(end+1) = [];
	else
	sys(end+1) = var:;
	end
	out = true;
	else
	out = false;
	end

public Integer pop()
if (fifo.isEmpty())
return null;
else
return remove(size()-1);
end
end
	if empty(sys)
	element = [];
	else
	element = sys(1);
	sys(1) = [];
	assignin('caller','sys',sys);
	end
	system = sys;

Table 6.1: Development of the three FIFO functions in Java and MATLAB.

- how to learn a simple model with RALib;
- how to deal with the learning algorithm, modifying the characteristic parameters to obtain the desired output (e.g., how to define data types, theories, equivalence check, ...);
- that the learning process can be correctly merged with the MATLAB-Java interface.

The developed code for the algorithm works and follows the operation of the SL* algorithm. No assumptions have been made and the final models fully represent the case study. Only three counterexamples have been necessary to conclude the learning process. A complete comprehension of the library and of the algorithm have been gained. Different definitions of nodes have been tested and we concluded that the proposed one is the one that best fits this case study. The interaction between the algorithm and an external SUL has been shown to work both on the native language and on an external SUL. It

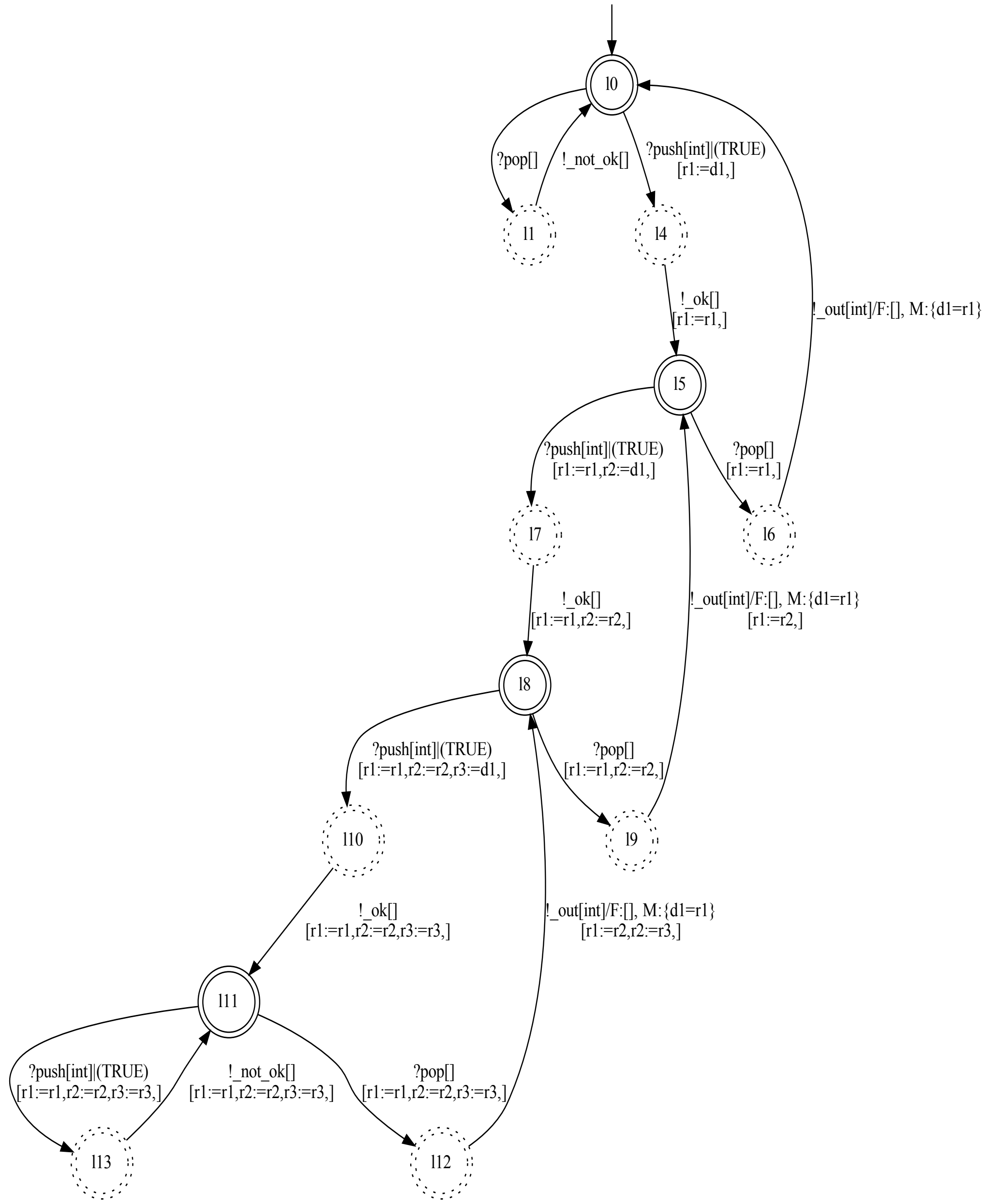


Figure 6.1: Complete model of the canonical FIFO system developed in Java.

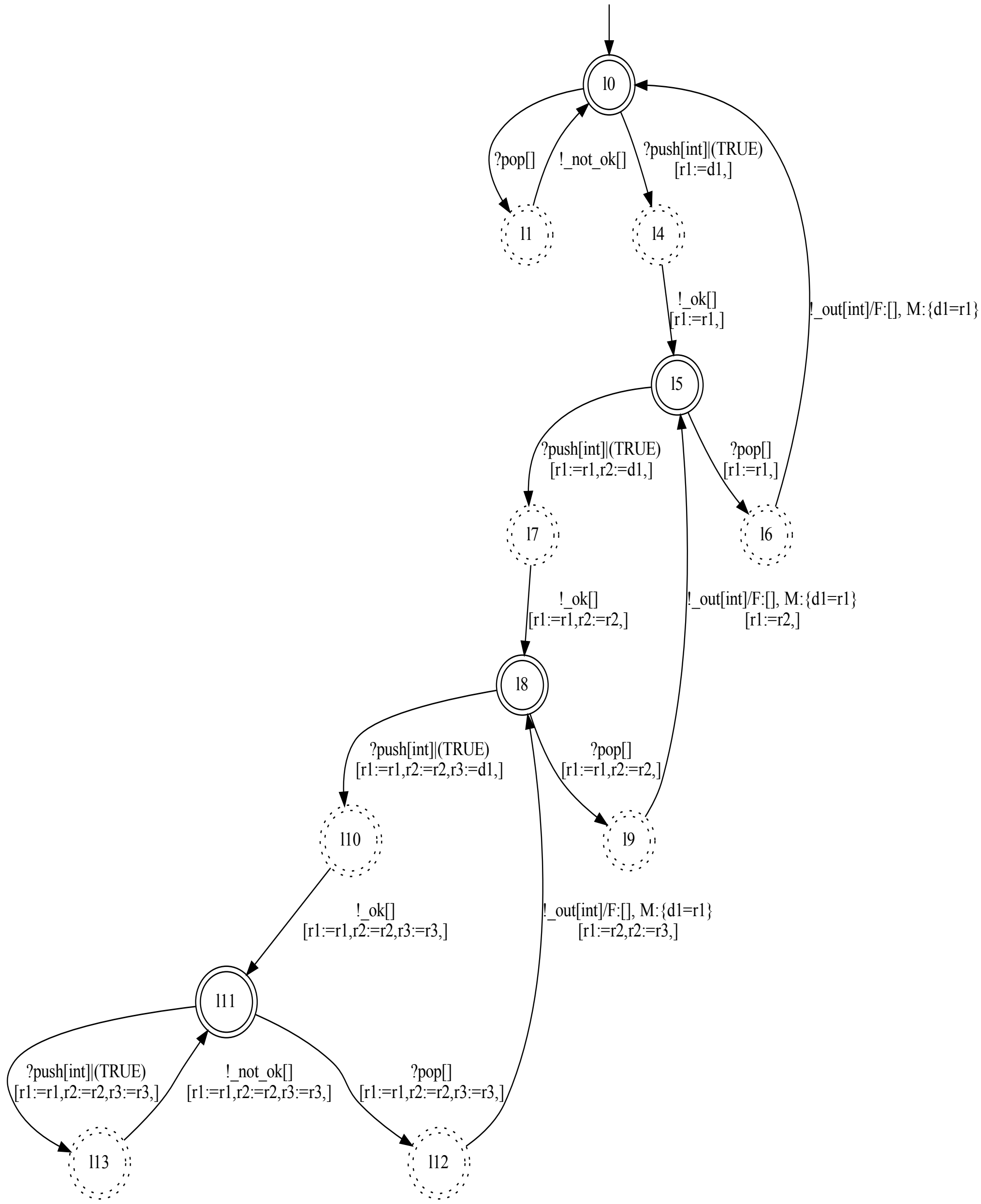


Figure 6.2: Complete model of the canonical FIFO system developed in MATLAB.

provides a more concrete demonstration on the correctness of the MATLAB-Java interface.

The graphs (Figure 6.1, 6.2) are now described to better understand the results. We are only going to present one of them since they are similar. It is important to provide the definition of state, adapted to this specific experiment. Every accepted node is characterized by a state of the vector with different capacity sizes. In our case study the system has maximum dimension of three elements, so we have 4 nodes. `l0` is the empty vector, `l1` has one position occupied, `l2` two position used and `l3` when the vector is full. The graph starts from the node `l0`. This location defines the empty vector. More generally, it is the state characterized as in the initialization of the program. We largely discussed that the FIFO system presents two functions: push and pop. These two functions permit the change in state of the system.

The model is in the form of a Mealy machine, where the output function is presented directly on the transition. Each transition arrow is divided in two parts to keep the labelling clear and readable. An intermediate fictitious states, drawn as double dotted circles, is used to separate the input and the output part on the transition. From the starting node to the intermediate node is highlighted the input function. After it, the output corresponding to that transition is described. If both labels have been in the same point of the transition, it would have become complex to understand.

The two possible requests to the system are in the form:

- $?pop[\ ]$;
- $?push[int]|(TRUE)[r1 := d1,]$.

In the initial state, the pop request can not be accomplished because the vector is empty. This brings to a nonacceptable output function ($!\_not\_ok[\ ]$). The state does not change, so the requested function return to the initial state `l0`. The push request can be performed because there are still available spaces in the vector. The output function is $!\_ok\ \ [r1 := d1,]$. This output function means that the transition is accepted ($!\_ok$) and the data value of the push function `d1` is saved in the first element of the register `r1`. The characterization of the state changes since the capacity of the system changes. The final state for this transition is `l5`. Iteratively, at every state, both functions are experimented. For both the states `l5` and `l8`, where the vector is not full and not empty, every transition is accepted ($!\_ok\ \ [...]$). The push functions bring the state to an higher level (e.g., from one used element `l5`, to two used elements `l8`, ...). The registers need to be updated with the new data value $!\_ok\ [r1 := r1, r2 := r2]$. The pop functions increment the space of the state (e.g., from `l5`, with one element, to `l0`, with no elements), printing as output the first element in the vector, or rather in the register `r1` (!_$out[int]$, $M : \{d1 = r1\}$). It updates registers ($r1 := r2$) moving every element one position ahead in the register's array. The last element of the model, `l11` represents the case in which the system is full. A push request can not be accepted (!_$not_ok[\ ]$ $[r1 := r1, r2 := r2, r3 := r3,]$). The pop function can be accepted and returns a feasible output saved in the register.

Some important data regarding the learning process are presented in table 6.2

	FIFO Java	FIFO MATLAB
Resets [ ]	1120	1367
Inputs [ ]	10250	10970
Locations [ ]	12	12
Transitions [ ]	16	16
Input Locations [ ]	1	1
Input Transitions [ ]	8	8
Build time [s]	27	32

Table 6.2: Canonical FIFO learning data summary table.

From this table, it is possible to understand that the two models have the same number of locations, transitions, input locations, and input transitions. Further conclusions can be drawn looking at the effective model and to the Figure 6.1 and 6.2. Intuitively the learning process of the Java FIFO experiment is faster since no translation between two different languages is needed. Experimental data show that this assumption is true.

6.3 Learning FIFO with constraint

A slightly more complex step consists in the development of a FIFO system introducing the previously presented constraints on the data values. The SUL used during this experiment is the same one as the one proposed as benchmark example in Section 2.4. To propose the same exact example, the studied system only has one acceptable position inside the vector that defines the system. The constraint on the data value is given by the parameter 2 on the push function. If the value 2 is pushed into the system, the transition does not produce an acceptable output function. Value 2 is nothing more than a random value chosen by way of example. For these experiments, the SUL has again been tested both in Java and MATLAB. Since the results are exactly the same, we will only present the MATLAB version since it is the most relevant to our purpose.

Some lines of code have been added to the SUL definition to implement the new constraint. In the push function, the following check loop has been introduced:

Listing 6.3: Push function with constraint in MATLAB

```
if (la < capacity)
    if (isempty(varargin))
        sys(end+1) = {[]};
        out = true;
    elseif (isequal(val, 2))
        out = false;
    else %input argument is not a cell
        sys(end+1) = varargin{:};
        out = true;
    end
else
    out = false;
    disp 'The system is full'
end
```

Looking at this code, it is now clear that not all data values in the push function bring to the same state. Value 2 is not acceptable and returns a false statement. The same `push` function, with different data values produces different outputs. All the output cases need to be studied. In the specific example we know that the transition function can be `accepted` or `rejected`. Constants need therefore to be introduced. In Section 2.4 it is said that the learning algorithm should check every function with all the constants and a single value different from every other constant. In this example, only one constant is needed to define both the possible states. The algorithm is going to check each function with `(c1 == d1)` and `(c1 != d1)` data values. The constant parameter c1 is defined as follows:

Listing 6.4: Constant definition for FIFO with constraint

```
Constants constants = new Constants();
ConstantGenerator cGen = new
   SymbolicDataValueGenerator.ConstantGenerator();
Constant c1 = cGen.next(FIFOSUL.INT_TYPE);
constants.put(c1, new DataValue(FIFOSUL.INT_TYPE, 2));
```

This definition allows the system to understand which values to check. Without these few lines, the system wouldn't have managed to converge.

The final output of this experiment reflects therefore the expected one presented before (Figure 2.10).

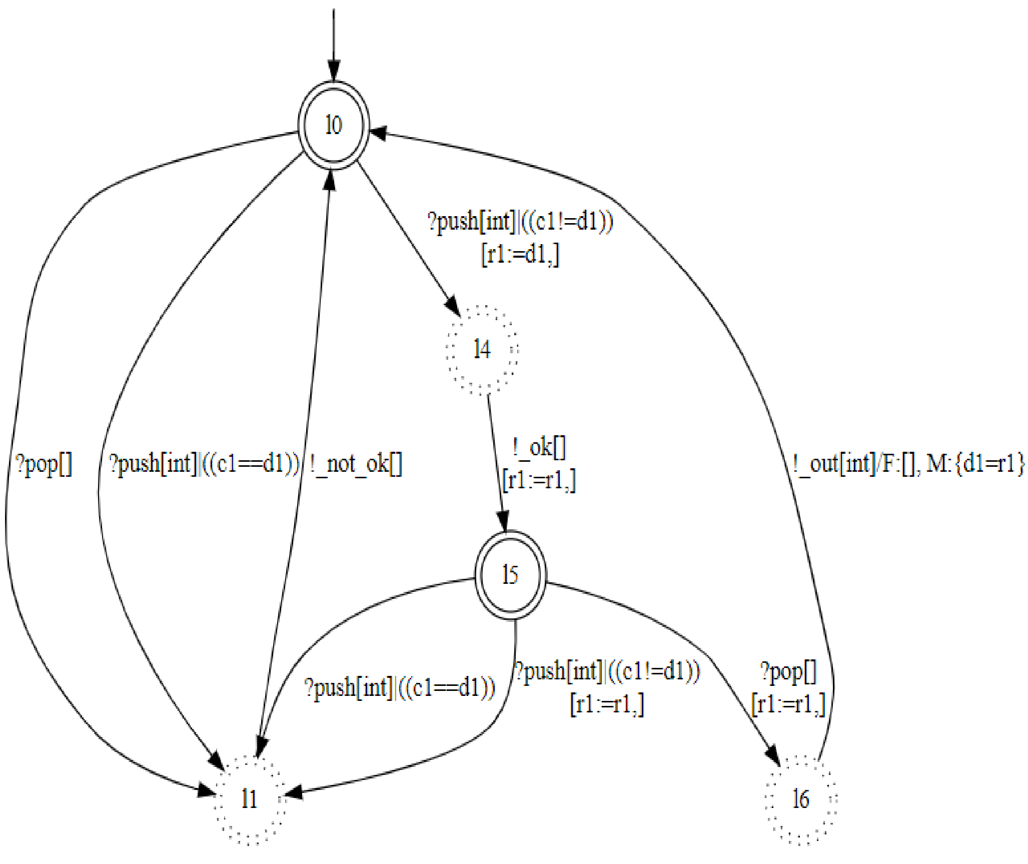


Figure 6.3: Model FIFO system with constraints.

It is immediately clear that what is represented in the graph coincides with what is expected and described in Chapter 2. Figure 2.10 is isomorphic to Figure 6.3. This proves that the model not only is correct, but also respects the desired outcome.

The initial state is `l0`. It is the one only reached from the initialization function. The three studied actions are:

- `?pop[];`
- `?push[int]|(c1==d1) [r1:=d1,];`
- `?push[int]|(c1!=d1) [r1:=d1,].`

Only the function `?push[int]|(c1!=d1) [r1!=d1,]` bring to an acceptable state. After this action the vector contains one element. The new node is here called `l5`. Both push functions are rejected because the system is full. The pop function outputs the desired element from the vector. This structure is the most simplified one, for larger capacities, the model iterates the fundamental block, enlarging the graph. It requires much more computational power to obtain a larger model. It is due to the complexity of the system and the growth in number of queries and counterexamples.

The learning process data are presented in table 6.3

	FIFO constraint MATLAB
Resets [ ]	42
Inputs [ ]	2741
Locations [ ]	7
Transitions [ ]	9
Input Locations [ ]	1
Input Transitions [ ]	6
Build time [s]	25

Table 6.3: FIFO constraint learning data summary table.

The complexity of this system is much higher than the simple FIFO system. It is possible to see that the building time is similar to the simple FIFO, but only one element of the array is considered against the three-element vector of the previous case. Since the two systems, although based on the same logic, present different complexities (number of states, transitions, or queries needed), the building times can not be compared. It is therefore not possible to draw a clear trend in the time consumption.

6.4 Learning LSM

Thanks to the experiments previously presented, we have full knowledge of how the system is supposed to learn and how to interact between the two platforms used. The same process used to learn the previous examples and everything that has been implemented in Section 6.1, 6.2, 6.3 has been used to create the learning algorithm for the SUL. The next logical step is the conjunction of the benchmark examples previously studied to start the learning process of the LSM.

Chapter 3 gave a full characterization of the LSM and much useful information can be retrieved. From the literature study [1], we know that the size of the learned algorithm, and consequently the alphabet size and the number of counterexamples can affect the efficiency and termination of the algorithm.

The first attempt at the LSM case study has been analyzed with the complete system and the algorithm did not manage to terminate with a concrete solution. The learning algorithm successfully starts, but the observation table grows without permitting even the production of the first hypothesis. After 10 hours of training, the process was stopped. In order to further understand and analyse the learning process, other experiments have been made, imposing progressively higher levels of abstraction. The first type of assumption made was regarding constant parameters and data types. Variables have been grouped under the same data type if they present the same exact behaviour in the learned system. Constant parameters have been retrieved looking inside the SUL MATLAB code.

A further step has been producing a simpler systems with fewer variables. Variables can not be removed randomly, but only merged if they present the same behaviour by studying the implemented MATLAB code. In this case, the number of states and variables has been manually reduced, creating a new smaller system. Though such an effort changes the implementation of the MATLAB code, the structure and logical behaviour of the system has been preserved.

The original system has been manipulated to exclude some variables highly dependent on each other. The new simplified SUL has been developed as a reduced LSM with the five variables that were considered most significant for the new system (`state, direction, bool4, bool5, bool6`) instead of the original sixteen. This type of manual abstraction should be avoided as most as possible since it can introduce other errors in the learned SUL, but in this case it was necessary to obtain a solution and analyse the learning algorithm.

To permit the algorithm to create a model, the characteristics specified before (Chapter 2) have to be coded, similarly to the previous examples. Those are presented here more in detail.

The five variables that compose the input symbol are characterized by different constant values that trigger the change in state. It would be possible to define a unique data type with all the constants. This will produce a growth in the number of queries. Creating different data types will not compromise the complexity of the learning algorithm, but will drastically reduce the required computation effort. All and only the relevant cases should be studied.

Looking at the LSM, it is possible to detect the constants parameters that trigger each variable. All variables that have the same parameters could be grouped under the same data type. To each data type are assigned constants and a specific theory.

Listing 6.5: Example code with constant parameters

```
if (direction == 1)
    direction = 'left';
elseif (direction == 2)
    direction = 'right';
else
    direction = 'none';
end

if (bool4 == 1)
    bool4 = true ;
else % elseif bool4 == 0
    bool4 = false;
end

if (bool5 == 1)
    bool5 = true;
else % elseif bool5 == 0
    bool5 = false;
end
```

Listing 6.5 shows an example code to describe the concept of abstraction with constant parameters. The listing shows how variables `bool4` and `bool5` are triggered from the same constant parameter (1). The two variables can therefore be classified under the same data-type and theory. Variable `direction` instead produces a change in state depending on both data values 1 and 2. A new data type has to be created to consider all exit states from `direction`. While this new data type could also have been used for variables `bool4` and `bool5`, it would have led to a longer experiment given the higher number of studied cases, higher number of queries, and therefore more computational effort to solve the same system. It should be avoided as much as possible since the biggest problem encountered until now has been related to the computation effort needed to solve the original problem.

In Listing 6.6 it is possible to see how different data types, theories, and teachers have been defined.

Listing 6.6: Definition of different data-types and theories

```
//definition of data types
public static final DataType DIR_TYPE = new
   DataType("dir", Integer.class);
public static final DataType BOOLEAN_TYPE = new
   DataType("bool", Integer.class);

//definition theories
IntegerEqualityTheory dirTheory = new
   IntegerEqualityTheory(DIR_TYPE);
IntegerEqualityTheory boolTheory = new
   IntegerEqualityTheory(BOOLEAN_TYPE);

//definition map that connects types and theories
Map<DataType, Theory> theories = new LinkedHashMap<>();
theories.put(DIR_TYPE, dirTheory);
theories.put(BOOLEAN_TYPE, boolTheory);
```

The two proposed categories are called `DIR_TYPE` and `BOOLEAN_TYPE`. Each data type has been connected to a specific integer equality theory (`dirTheory`, `boolTheory`). The resulting teachers are therefore `(DIR_TYPE, dirTheory)` and `(BOOLEAN_TYPE, boolTheory)`.

The different data types defined above are differentiated by the constant parameters that the corresponding assigned variables need. Constant parameters are defined unequivocally in Listing 6.7, which shows the usage of different data-types and constants.

Listing 6.7: Definition of constants

```
//definition of constants

Constants constants = new Constants();
// make sure to define constants for every type
    //constant for DIR_TYPE
   ConstantGenerator cGen = new
      SymbolicDataValueGenerator.ConstantGenerator();
   Constant c1 = cGen.next(DIR_TYPE);
   Constant c2 = cGen.next(DIR_TYPE);
   constants.put(c1, new DataValue(DIR_TYPE, 0));
   constants.put(c2, new DataValue(DIR_TYPE, 1));

   //constant for BOOLEAN_TYPE
   Constant cboolFalse = cGen.next(BOOLEAN_TYPE);
   constants.put(cboolFalse, new
      DataValue(BOOLEAN_TYPE, 0));
```

If only `BOOLEAN_TYPE` with `cboolFalse` as the only constant was defined, the variables that need more constants would have not been fully studied. Contrary, defining a single state `DIR_TYPE` with more constant values like `c1` and `c2`, will bring to an extensive study with too many not needed cases taken into consideration.

Other introduced abstractions are related to the minimization of parameters. In Section 5.2.2 it has been said that the random walk method would have been used for the counterexample processing phase. Other constraints on the maximum number of parameters to be studied and on the maximum number of iterations of the learning process have been imposed.

Listing 6.8: Definition of the main function

```
public static void main(String[] args) throws
    IllegalArgumentException, IllegalStateException,
    InterruptedException, ExecutionException,
    TimeoutException {
   bindParameters();
   testRandomly();
   runLearningExperiment(
         3 // test length bound
         );
```

```
}

// constraint maximum number of parameters
public static final int NUM_PARAM = 5;
```

After the imposed simplification, the number of computations is minimized and the algorithm can manage to create a solution. Figure 6.4 shows the ultimate learned model corresponding to the LSM as SUL. This graph is the answer to our problem statement. We therefore demonstrated that it is possible to learn a RA model from a MATLAB code within an automotive case study.

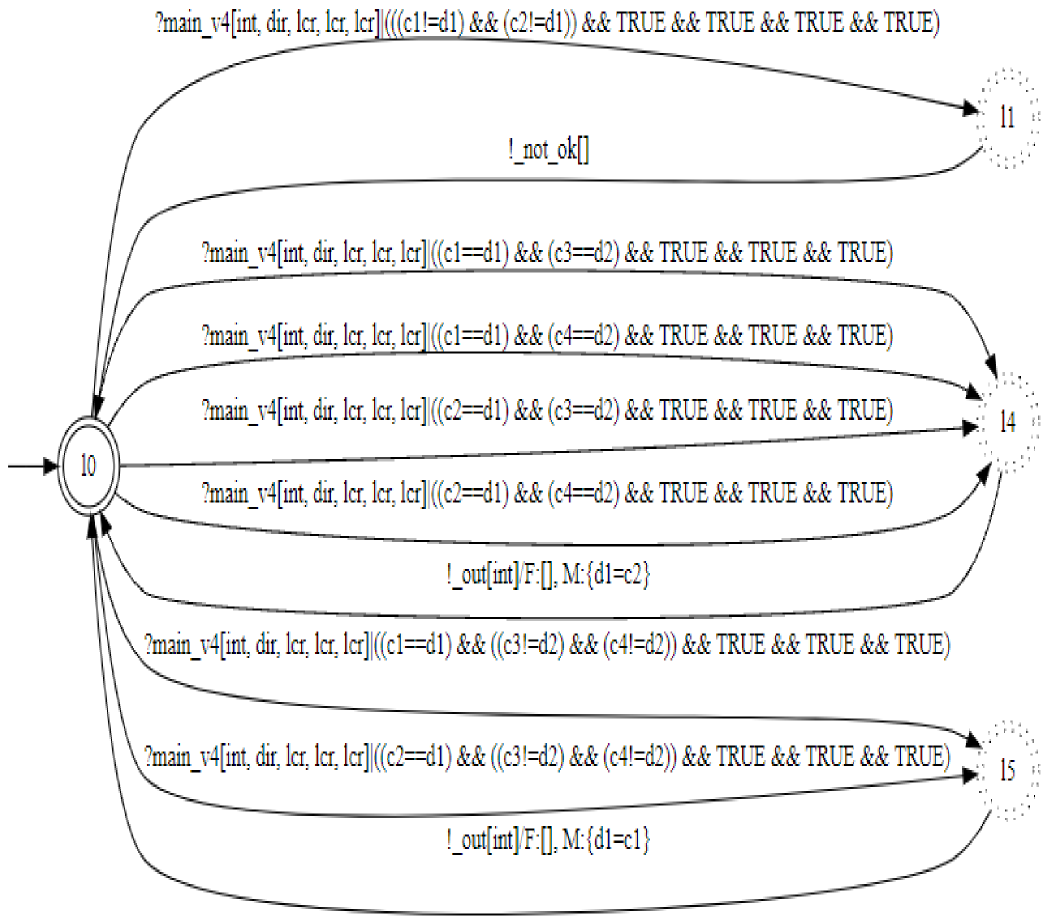


Figure 6.4: Model LSM with 5 variables.

Listing 6.9 shows the example code with the logical behaviour that corresponds to the outcome of the graph in Figure 6.4. Every part will be explained to provide a clear understanding of the model.

Listing 6.9: Example code for the behaviour of Figure 6.4

```
if state == 0
    state = 'stateA';
elseif state == 1
    state = 'stateB';
else
    state = non_acceptable_case
end
```

Every transition is fully determined by a function and the specifications of all its inputs. For example, the transition that brings to the intermediate state `l1` is composed as follow: `?main_v4[int, dir, lcr, lcr, lcr]|(((c1!=d1)&&(c2!=d1))&&TRUE&&TRUE&&TRUE&&TRUE)`. The first part `?main_v4[int, dir, lcr, lcr, lcr]|` describes how the function is generally composed. The second part `(((c1!=d1)&&(c2!=d1))&&TRUE&&TRUE&&TRUE&&TRUE)` specifies the designations on the input parameters. In this example the the first parameter d1 is fixed to a value different from both `c1` and `c2`. All other variables are said to be `TRUE`, so are not bounded to any constraint parameters.

Having a closer look at Listing 6.9 the variable `state` (d1) can only accept values with value 0 or 1. In all other cases, it will bring to a nonacceptable output function. Therefore, the imposed constraints `((((c1!=d1)&&(c2!=d1))&&TRUE&&TRUE&&TRUE&&TRUE))` will produce a nonacceptable output, although the final state is still belonging to the set of acceptable states `l0`.

All accepted states have been automatically merged in a single node by the algorithm, since those are characterized by the same values. It would have been possible to provide a different definition of state code wise, that would have therefore permitted a differentiation of different states also in the graph. For example, it would have been possible to use the variable `state` as discriminating parameter for the node definition. Also all other five original variables defining the state could have been used to produce a more complete graph, although the provided information would be the same. It will not add any value in the comprehension of the model and will raise the computational effort needed to build the hypothesis.

The second set of functions that converges to the intermediate state l4, considers constant values also for the second parameter `d2` (e.g., `((c1==d1)&&(c3==d2)&&TRUE&&TRUE&&TRUE)`). This specific example assigns the first data value `d1` to `c1` and `d2` to `c3`. The output function is characterized by an integer value that reassigns the parameter `d1` with a new value `c2`. It will lead to a change in the value of the variable `state`. The same process can also be used to understand the third and last set of functions that redefines `d1` with value `c1`. Using the symbolic Nerode equivalence and the subsequent theorem, as defined in Definitions 13 and 14, all equivalent states have been merged reducing the size if the final learned model.

With respect to the objective of this thesis, it has been shown that it is possible to use SL^* to learn a model of a part of the LSM, as presented in Figure 6.4. However, scaling this approach requires further research to improve the algorithms so that it will be possible to learn systems as large as the original LSM. It is possible to make further assumptions, but it is not advisable since those are error prone.

The key point that we need to gather from this example is that it is possible to learn a model starting from an external SUL implemented in MATLAB. Assumptions should be avoided as much as possible since they can introduce errors and some states can be missed. We should at the same time be aware that assumptions are needed to ensure the convergence of the algorithm. Previous experiments ([1]) with the large LSM managed to learn only thanks to strong abstractions, reducing in this way the size of the alphabet. We faced some problems in learning a bigger model without further abstractions due to the alphabet size. In that case, stronger assumptions need to be used. Theoretically, if the assumptions are right, the resulting automaton should present all states. Assumptions reduce the number of combinations of input variables while the possibility of a transition from one state to another is preserved.

The proposed abstraction manages to produce a model that represents the system. Nevertheless, this is not an ideal solution. An optimal implementation would be represented by a learning system that interacts with the SUL as a black box. Since we used the knowledge gained from analyzing the system, care has to be taken when scaling this approach as this might possibly introduce errors.

Once again, the characteristics of the learning process are presented in table 6.4.

	Simplified LSM
Resets [ ]	6355
Inputs [ ]	13280
Locations [ ]	6
Transitions [ ]	10
Input Locations [ ]	1
Input Transitions [ ]	7
Build time [s]	99s (1m 39s)

Table 6.4: Simplified LSM learning data summary table.

The computational effort of the system, immediately intelligible from the number of inputs and resets needed to learn the model, corresponds to a rise in the amount of time needed to end the full process. The rise is not linearly proportional to the corresponding parameters in the previous examples. The data used in these experiments are too few to draw any further conclusions.

Chapter 7

Conclusions

The scope of this thesis was to demonstrate that it is possible to learn a model from MATLAB code characterising a SUL. Starting from the previous study in [27], it was hypothesized that extended finite state automata formalisms like register automata could help address some of the challenges encountered. The usage of RALib was the first impactful decision taken. It is a library that implements an automata learning algorithm called SL*. It is an improvement of the more commonly used L* algorithm. SL* differs from it because it is based on register automata.

A structured methodology has been developed to create a code that can learn a model starting from MATLAB. The structure of the experiments has been thought to provide a solution and verifying every step. The developed methodology can be used also for future work if adapted to specific study cases. Developing the code was a drawback since the library is *quite recent and so it took some time to finish up the learning examples to retrieve the results*. In the end, all experiments have been developed and worked smoothly. Some assumptions and simplifications have to be made in order to let the algorithm produce the proposed solution of the SUL. Those manipulations of the code let us state that the proposed algorithm still is not the optimal solution for such big problems. Specifically, register automata can have better performance in the case in which data saved into registers are also used as output values. This is the case of the presented FIFO system. The dimension of the studied system was the main bottleneck in this project.

7.1 Future works

This project was born as a possible solution to address the manual creation of formal models for verifying AD algorithms. AD algorithms, although they can be reduced to smaller subsystems, are large. A personal thought of the author is that the current proposed solution is still far behind the major problem of learning a full model of such big systems without assumptions. Due to the breadth of the problem, the initial goals have been met only partially. The original system, without making any abstractions and assumptions was not learned. In this section, we will focus on some of the remaining issues that should be addressed in future work. Papers [27, 62, 63] and expert advices made the author gain more knowledge on possible future development of the code. New algorithms can be tested. Some of them can be for example, TTT [22] or discrimination tree developed from LearnLib [19]. It is still highly probable that also these new algorithms do not produce an ideal solution. In the case in which the state of the art will be redefined, deep studies of the proposed solutions can be done. Some features that could possibly improve the solution are:

- manage to learn correlations between data, reducing the size of the verification problem;
- treat the system as a black box;
- more complex minimization algorithm;
- reducing the size of counterexamples with faster methods.

Using a grey box learning tool could also improve the feasibility of the solution since will remove some steps in the learning process. The recent paper [64] proposes a grey box learning model-based on RALib. It claims that this new methodology results in almost two orders of magnitude improvement in terms of the numbers of inputs sent to the software system.

Starting from the proposed code, other assumptions can be made. Manually defining the relation between input variables can be proposed as future work. Adding further assumptions can help obtaining the solution of the full model, but bring us away from the real aim of the project to treat the system as a black box.

The paper [65] proposes a method for minimizing a Moore finite-state machine, such as our SUL, by merging two states. This method proposes some conditions that guarantee the identical operation and deterministic behavior of the transformed FA obtained by merging two states. This method can help in the success of the problem statement.

7.2 Use of automaton learning for automotive ECU testing

AD is a highly studied topic in recent time. Many researches and papers try to ensure software quality with various techniques. We should be aware that the path we decided to follow is not the only one and others can bring similar results. For this reason, we believe it is also useful to present different methodologies that present promising results to solve the same challenge.

The following papers still rely on automata learning, although with some changes in the practical application or the setup. A general survey to compare several known algorithms for learning DFA via membership and equivalence queries is presented in [66]. This presentation is supposed to help the reader to gain the first needed introduction to the general implementation of automata learning [67].

A research group at Kungliga Tekniska Högskolan (KTH) has developed results and papers for a fully automated testing tool for testing of Electronic control units (ECU) using automata learning [3]. Papers such as [12, 29] describe various techniques to learn a model using active ML. These theories can then be analysed to predict unseen bugs.

The authors of [68] proposed an effective strategy to directly verify safety-critical components in cyber-physical systems in the automotive field. It focuses on a Learning-based testing (LBT) learning-based testing as a paradigm to automatically test the adherence of the software requirements of the SUL. Papers [69, 70], shown that LBT can model a SUL, finding bugs and existing errors in the SUL code. This testing strategy combines model checking together with ML algorithms [29, 66, 71]. Automata learning incrementally learns a model of the SUL, using test cases as queries. Different learning algorithms have been evaluated in various papers ([72, 73]) such as Algebraic parameter estimation, IKL([74]) (an incremental learning algorithm for Kripke structures) and L* Mealy [75].

The case study proposed in this paper is related to AD, specifically the verification of multi-vehicle platoons*. It defines a unique setup to construct

* A vehicle platoon (or road train) is a collection of vehicles that coordinate and collaborate to reach goals such as travelling to a certain destination, while also improving e.g., safety, fuel economy and driver comfort [76]

the model and check its quality and compliance with the requirements. The automaton learning algorithm is based on regular inference [29] through queries to a black box system [34] as in our project. Only a partial model of the SUL is verified at each time with the requirement specifications to test the requirements. Testing techniques based on incomplete models are suggested in [77, 78]. Those enable the model-checking even of reduced parts of a complete model according to changes in the software. Only the changed part is in this way studied. Some promising results have been achieved trying to scale up LBT to test Cooperative open cyber-physical systems-of-systems (CO-CPS).

Paper [79] deals with automotive applications and the lack of precise formal modelling requirements. It proposes a method to transform the system requirements from their traditionally written form into a more formal notation. Also, [80] covers some peculiar aspects of the lack of formal specifications and its causes focusing on the automotive software industry. Paper [81] proposes a large study of the behavioural requirements for an automotive case study for Bosch. They used Real Time Specification Pattern System (RTSP), presented in [82] as extension of Specification pattern system (SPS) presented in [83].

In [76] a general methodology to estimate safety-related parameter values of CO-CPS is presented. The same vehicle platoon is used as a case study using a renovated distributed protocol for coordinated emergency braking. The estimation methodology is based on learning-based testing. This method can be exploited to quantitatively analyse the system behaviour, especially for the properties that impact safety. It is based on a general LBT method, as proposed in [69] and in [68].

Applying LBT to spatio-temporal requirements on autonomous systems is also presented in [84] bringing to a Spatio temporal automata (STA). As in [68] the SUL model is learned through automata learning. Static analysis is instead used to control the consistency of the software requirements against the just created model. Dividing these two steps and using different tools can improve the automaton level of the general control process. It has been proved that important results can be achieved with this method. It is said that on an SUL with a reasonably low test latency, test throughput of the order of hundreds of thousands of test cases per hour is achievable using hardware acceleration.

Other papers propose alternative approaches to this problem [85, 86, 87, 88]. Additional improvements can be implemented on the usage of graph queries. In [89] a new model-checking approach based on STA [84] is presented. This work is promising and differs from others since it introduces graph queries in the model-checking phase to falsify the STA requirements [90]. The model-checking problem can be therefore reduced to a simple query matching problem. Other papers like [91] instead reference to the open-source NuSMV tool [92] as a model-checker. Other model-checker possibilities can be found in [93]. Paper [91] references to the previous works on LBT demonstrates that this approach, with its various aspects, is applicable to requirements testing of low latency safety-critical embedded systems, such as can be found in the automotive sector. [94] deals with the effectiveness of the LBT testing strategy for automotive applications.

Other papers like [95, 96] deal with autonomous driving and ensure the process of the development requirements. In paper [95] is proposed an analysis of a collection of automotive requirements by using formal methods. A model-based approach for the software design process is proposed. It relies on formal verification techniques to develop a design solution for system requirements and to verify the developed software.

In [96] is presented a Statistical Model Checking (SMC) method proposed as a possible solution for verifying the safety of autonomous systems and various algorithms. It is applied on a Heuristic Autonomous Intersection Management (HAIM) algorithm. The developed verification routine can be adopted for other conflict point-based autonomous intersection management algorithms as well.

References

[1] Y. Selvaraj, A. Farooqui, G. Panahandeh, and M. Fabian, "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*, 2020. doi: 10.1145/3417990.3421262 pp. 1–10. [Online]. Available: https://www.researchgate.net/publication/344927821_Automatically_Learning_Formal_Models_An_Industrial_Case_from_Autonomous_Driving_Development

[2] "Ansi/ul, ansi/ul 4600 - standard for evaluation of autonomous products. presenting the standard for safety for the evaluation of autonomous vehicles and other products," https://ul.org/UL4600, accessed: 2021-02-26.

[3] K. Meinke, "Active machine learning to test autonomous driving," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2021. doi: 10.1109/ICSTW52544.2021.00055 pp. 286–286. [Online]. Available: https://ieeexplore.ieee.org/document/9440164

[4] "SAE J3016," https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic, accessed: 2021-03-05.

[5] I. Barabás, A. Todoruţ, N. Cordoş, and A. Molea, "Current challenges in autonomous driving," *IOP Conference Series: Materials Science and Engineering*, vol. 252, p. 012096, oct 2017. doi: 10.1088/1757-899x/252/1/012096. [Online]. Available: https://doi.org/10.1088/1757-899x/252/1/012096

[6] E. Xidias, P. Zacharia, and A. Nearchou, "Path planning and scheduling for a fleet of autonomous vehicles," *Robotica*, vol. 34, no. 10, p. 2257, 2016. doi: 10.1017/S0263574714002872. [Online]. Available: https://doi.org/10.1017/S0263574714002872

[7] "Why don't we have self-driving cars yet?" https://www.youtube.com/watch?v=qf6VrDZ04EQ, accessed: 2021-02-26.

[8] "Road vehicles - functional safety iso26262," https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en, accessed: 2021-02-15.

[9] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016. doi: 10.1016/j.tra.2016.09.010. [Online]. Available: https://doi.org/10.1016/j.tra.2016.09.010

[10] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016. doi: 10.4271/2016-01-0128. [Online]. Available: https://saemobilus.sae.org/content/2016-01-0128/

[11] "Ainomaly fengco mxl tech pitch 2021," https://www.youtube.com/watch?v=JiuwnqDU-6c&t, accessed: 2021-02-18.

[12] A. Bennaceur and K. Meinke, "Machine learning for software analysis: Models, methods, and applications," in *Machine Learning for Dynamic Software Analysis: Potentials and Limits*. Springer, 2018, pp. 3–49. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-96562-8_1

[13] "Formal methods, carnegie mellon university, spring 1998, michael collins," https://users.ece.cmu.edu/~koopman/des_s99/formal_methods/, accessed: 2021-03-09.

[14] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software engineering journal*, vol. 8, no. 4, pp. 189–209, 1993. doi: 10.1049/sej.1993.0025. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/sej.1993.0025

[15] A. Zita, S. Mohajerani, and M. Fabian, "Application of formal verification to the lane change module of an autonomous vehicle," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*. IEEE, 2017. doi: 10.1109/COASE.2017.8256223 pp. 932–937. [Online]. Available: https://ieeexplore.ieee.org/document/8256223

[16] K. S. Weldemariam, "Using formal methods for building more reliable and secure e-voting systems," Ph.D. dissertation, University of Trento, 2010. [Online]. Available: http://eprints-phd.biblio.unitn.it/253/1/Thesis.pdf

[17] Z. Jun, "Overview of formal methods in software reengineering [j]," *Science Mosaic*, vol. 5, 2009. [Online]. Available: https://en.cnki.com.cn/Article_en/CJFDTotal-KJIG200905003.html

[18] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008. ISBN 026202649X. [Online]. Available: https://dl.acm.org/doi/book/10.5555/1373322

[19] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *International Conference on Computer Aided Verification*. Springer, 2015. doi: $10.1007/978-3-319-21690-4\_32$ pp. 487–495. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-319-21690-4_32

[20] F. D. Aarts, "Tomte: bridging the gap between active learning and real-world systems," Ph.D. dissertation, [Sl: sn], 2014. [Online]. Available: https://repository.ubn.ru.nl/handle/2066/130428

[21] S. Cassel, F. Howar, and B. Jonsson, "Ralib: A learnlib extension for inferring efsms," 2015. [Online]. Available: https://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf

[22] M. Isberner, F. Howar, and B. Steffen, "The ttt algorithm: a redundancy-free approach to active automata learning," in *International Conference on Runtime Verification*. Springer, 2014. doi: $10.1007/978-3-319-11164-3\_26$ pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-319-11164-3_26

[23] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994. doi: 10.1016/0304-3975(94)90010-8. [Online]. Available: https://doi.org/10.1016/0304-3975(94)90010-8

[24] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen, "A succinct canonical register automaton model," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2011. doi: $10.1007/978-3-642-24372-1\_26$ pp. 366–380. [Online]. Available: https://doi.org/10.1007/978-3-642-24372-1_26

[25] A. Farooqui and M. Fabian, "Synthesis of supervisors for unknown plant models using active learning," in *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2019. doi: 10.1109/COASE.2019.8843177 pp. 502–508. [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/8827189/proceeding

[26] M. A. Sindhu and K. Meinke, "Ids: An incremental learning algorithm for finite automata," 2012. [Online]. Available: https://arxiv.org/abs/1206.2691

[27] Y. Selvaraj, "On provably correct decision-making for automated driving," *research.chalmers.se*, 2020. [Online]. Available: $https://research.chalmers.se/publication/518961/file/518961_Fulltext.pdf$

[28] S. Giorgio, "Automata, languages and computation chapter 2 : Finite automata," University Lecture, 2020. [Online]. Available: https://elearning.dei.unipd.it/course/view.php?id=9254

[29] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010. [Online]. Available: https://doi.org/10.1017/CBO9781139194655

[30] S. Cassel, "Learning component behavior from tests: theory and algorithms for automata with data," Ph.D. dissertation, Acta Universitatis Upsaliensis. ISBN 978-91-554-9395-0 2015. [Online]. Available: http://uu.diva-portal.org/smash/record.jsf?pid=diva2%3A865360&dswid=7096

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Automata theory, languages, and computation," *International Edition*, vol. 24, no. 2, 2006.

[32] "Programming in tabled prolog," https://www3.cs.stonybrook.edu/~warren/xsbbook/book.html, accessed: 2021-03-09.

[33] B. Holdsworth and C. Woods, *Digital logic design*. Elsevier, 2002. [Online]. Available: https://books.google.it/books?hl=en&lr=&id=o7enSwSVvgYC&oi=fnd&pg=PP1&dq=digital+logic+design+holdsworth+&ots=zEBo2wkefs&sig=_wxwoAC8l4EHpNY41dhM_0lDHd8&redir_esc=y#v=onepage&q=digital%20logic%20design%20holdsworth&f=false

[34] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987. doi: 10.1016/0890-5401(87)90052-6. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6

[35] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. Springer, 2018, vol. 10. [Online]. Available: https://link.springer.com/book/10.1007%2F978-3-319-10575-8

[36] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines," *Formal Aspects of Computing*, vol. 28, no. 2, pp. 233–263, 2016. doi: 10.1007/s00165-016-0355-5. [Online]. Available: https://doi.org/10.1007/s00165-016-0355-5

[37] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation, 2nd edition," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/568438.568455?casa_token=4wzts1T20UkAAAAA:dqBHFi7MwlxNcLLftnquyzH-Mr5ZQTDXfhgjINpnv6vx-oSHIYhJ00BUCD_R65MnJ98U0B-zOnY

[38] A. Nerode, "Linear automaton transformations," *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958. doi: /10.2307/2033204. [Online]. Available: https://doi.org/10.2307/2033204

[39] F. Vaandrager and A. Midya, "A myhill-nerode theorem for register automata and symbolic trace languages," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2020, pp. 43–63. [Online]. Available: https://arxiv.org/abs/2007.03540

[40] E. M. Gold, "Complexity of automaton identification from given data," *Information and control*, vol. 37, no. 3, pp. 302–320, 1978. doi: 10.1016/S0019-9958(78)90562-4. [Online]. Available: https://doi.org/10.1016/S0019-9958(78)90562-4

[41] H. Raffelt, B. Steffen, and T. Berg, "Learnlib: A library for automata learning and experimentation," in *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, 2005. doi: 10.1145/1081180.1081189 pp. 62–71. [Online]. Available: https://dl.acm.org/doi/10.1145/1081180.1081189

[42] F. B. Khendek, S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on software engineering*, vol. 17, no. 591-603, pp. 10–1109, 1991. doi: $10.1007/978-0-387-35567-2\_15$. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-0-387-35567-2_15

[43] M. Dorofeeva and I. Koufareva, "Novel modification of the w-method," *Bulletin of the Novosibirsk Computing Center. Series: Computer Science*, vol. 2002, pp. 69–80, 2002. [Online]. Available: https://nccbulletin.ru/files/article/dorofeeva1.pdf

[44] M. Isberner, F. Howar, and B. Steffen, "Learning register automata: from languages to program structures," *Machine Learning*, vol. 96, no. 1-2, pp. 65–98, 2014. doi: 10.1007/s10994-013-5419-7. [Online]. Available: https://doi.org/10.1007/s10994-013-5419-7

[45] S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Learning extended finite state machines," in *International Conference on Software Engineering and Formal Methods*. Springer, 2014. doi: $10.1007/978-3-319-10431-7\_18$ pp. 250–264. [Online]. Available: https://doi.org/10.1007/978-3-319-10431-7_18

[46] F. Aarts, F. Howar, H. Kuppens, and F. Vaandrager, "Algorithms for inferring register automata," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

doi: $10.1007/978-3-662-45234-9\_15$. ISBN 978-3-662-45234-9 pp. 202–219. [Online]. Available: https://doi.org/10.1007/978-3-662-45234-9_15

[47] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015. doi: $10.1007\%2F978-3-319-21690-4\_32$. ISBN 978-3-319-21690-4 pp. 487–495. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-319-21690-4_32

[48] J. Meijer and J. van de Pol, "Sound black-box checking in the learnlib," *Innovations in systems and software engineering*, vol. 15, no. 3, pp. 267–287, 2019. doi: 10.1007/s11334-019-00342-6. [Online]. Available: https://doi.org/10.1007/s11334-019-00342-6

[49] L. Ouedraogo, R. Kumar, R. Malik, and K. Akesson, "Nonblocking and safe control of discrete-event systems modeled as extended finite automata," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 3, pp. 560–569, 2011. doi: 10.1109/tase.2011.2124457. [Online]. Available: https://doi.org/10.1109/tase.2011.2124457

[50] V. Mitrana and R. Stiebe, "Extended finite automata over groups," *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 287–300, 2001. doi: 10.1016/S0166-218X(00)00200-6. [Online]. Available: https://doi.org/10.1016/S0166-218X(00)00200-6

[51] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July*. CSREA Press USA, 2013. ISBN 1-60132-243-7 pp. 67–73. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A677684&dswid=1681

[52] M. Merten, M. Isberner, F. Howar, B. Steffen, and T. Margaria, "Automated learning setups in automata learning," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2012. doi: $10.1007/978-3-642-34026-0\_44$ pp. 591–607. [Online]. Available: https://doi.org/10.1007/978-3-642-34026-0_44

[53] "Model a state machine for hdl code generation," https://it.mathworks.com/help/hdlcoder/ug/model-a-state-machine-for-hdl-code-generation.html, May 2021.

[54] "Calling matlab from java," https://it.mathworks.com/help/matlab/matlab-engine-api-for-java.html, May 2021.

[55] "Build java engine programs," https://it.mathworks.com/help/matlab/matlab_external/setup-environment.html, May 2021.

[56] "Java engine api summary," https://it.mathworks.com/help/matlab/matlab_external/java-api-summary.html, May 2021.

[57] "Execute matlab functions from java," https://it.mathworks.com/help/matlab/matlab_external/execute-matlab-functions-from-java.html, May 2021.

[58] "Class interruptedexception," https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html, July 2021.

[59] "Class executionexception," https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutionException.html, July 2021.

[60] M. Benedikt, C. Ley, and G. Puppis, "What you must remember when processing data words," in *Proceedings of AMW 2010*, vol. 619. CEUR-WS.org, 2010, pp. http–ceur. [Online]. Available: https://www.cs.ox.ac.uk/files/3092/amw-main.pdf

[61] M. Kaminski and N. Francez, "Finite-memory automata," *Theoretical Computer Science*, vol. 134, no. 2, pp. 329–363, 1994. doi: 10.1016/0304-3975(94)90242-9. [Online]. Available: https://doi.org/10.1016/0304-3975(94)90242-9

[62] Y. Selvaraj, W. Ahrendt, and M. Fabian, "Verification of decision making software in an autonomous vehicle: An industrial case study," in *International Workshop on Formal Methods for Industrial Critical Systems*. Springer, 2019. doi: $10.1007/978-3-030-27008-7\_9$ pp. 143–159. [Online]. Available: https://doi.org/10.1007/978-3-030-27008-7_9

[63] A. Farooqui, "On supervisor synthesis via active automata learning," *researchgate.net*, 2021. [Online]. Available: https://research.chalmers.se/en/publication/523934

[64] B. Garhewal, F. Vaandrager, F. Howar, T. Schrijvers, T. Lenaerts, and R. Smits, "Grey-box learning of register automata," in *International Conference on Integrated Formal Methods*. Springer, 2020. doi: $10.1007/978-3-030-63461-2\_2$ pp. 22–40. [Online]. Available: https://doi.org/10.1007/978-3-030-63461-2_2

[65] A. Klimovich and V. Solov'ev, "A method for minimizing moore finite-state machines by merging two states," *Journal of Computer and Systems Sciences International*, vol. 50, no. 6, pp. 907–920, 2011. doi: 10.1134/s1064230711040113. [Online]. Available: https://doi.org/10.1134/s1064230711040113

[66] J. L. Balcázar, J. Díaz, R. Gavalda, and O. Watanabe, "Algorithms for learning finite automata from queries: A unified view," in *Advances in Algorithms,*

Languages, and Complexity. Springer, 1997, pp. 53–72. [Online]. Available: https://doi.org/10.1007/978-1-4613-3394-4_2

[67] B. Sophia, "Learning-based testing of automotive ecus," 2016. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1065359&dswid=9156

[68] K. Meinke, "Learning-based testing of cyber-physical systems-of-systems: a platooning study," in *European Workshop on Performance Engineering*. Springer, 2017. doi: $10.1007/978-3-319-66583-2\_9$ pp. 135–151. [Online]. Available: https://doi.org/10.1007/978-3-319-66583-2_9

[69] K. Meinke and M. A. Sindhu, "Lbtest: a learning-based testing tool for reactive systems," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013. doi: 10.1109/icst.2013.62 pp. 447–454. [Online]. Available: https://doi.org/10.1109/icst.2013.62

[70] P. Nycander, "Learning-based testing of microservices: An exploratory case study using lbtest," 2015. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A847215&dswid=6524

[71] C. De La Higuera, "A bibliographical study of grammatical inference," *Pattern recognition*, vol. 38, no. 9, pp. 1332–1348, 2005. doi: 10.1016/j.patcog.2005.01.003. [Online]. Available: https://doi.org/10.1016/j.patcog.2005.01.003

[72] K. Meinke, F. Niu, and M. Sindhu, "Learning-based software testing: a tutorial," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2011. doi: $10.1007/978-3-642-34781-8\_16$ pp. 200–219. [Online]. Available: https://doi.org/10.1007/978-3-642-34781-8_16

[73] M. Sindhu, "Algorithms and tools for learning-based testing of reactive systems," Ph.D. dissertation, KTH Royal Institute of Technology, 2013. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A610371&dswid=-3564

[74] K. Meinke and M. A. Sindhu, "Incremental learning-based testing for reactive systems," in *International Conference on Tests and Proofs*. Springer, 2011. doi: $10.1007/978-3-642-21768-5\_11$ pp. 134–151. [Online]. Available: https://doi.org/10.1007/978-3-642-21768-5_11

[75] O. Niese, "An integrated approach to testing complex systems." Ph.D. dissertation, Technical University of Dortmund, Germany, 2003. [Online]. Available: http://eldorado.tu-dortmund.de/handle/2003/2545

[76] C. Bergenhem, K. Meinke, and F. Ström, "Quantitative safety analysis of a coordinated emergency brake protocol for vehicle platoons," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018. doi: $10.1007/978-3-030-03424-5\_26$ pp. 386–404. [Online]. Available: https://doi.org/10.1007/978-3-030-03424-5_26

[77] A. Groce, D. Peled, and M. Yannakakis, "Adaptive model checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002. doi: 10.1093/jigpal/jzl007 pp. 357–370. [Online]. Available: https://doi.org/10.1093/jigpal/jzl007

[78] A. Groce, A. Fern, M. Erwig, J. Pinto, T. Bauer, and A. Alipour, "Learning-based test programming for programmers," in *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2012. doi: $10.1007/978-3-642-34026-0\_42$ pp. 572–586. [Online]. Available: https://doi.org/10.1007/978-3-642-34026-0_42

[79] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas, "Reassessing the pattern-based approach for formalizing requirements in the automotive domain," in *2014 IEEE 22nd International Requirements Engineering Conference (RE)*. IEEE, 2014. doi: 10.1109/re.2014.6912296 pp. 444–450. [Online]. Available: https://doi.org/10.1109/re.2014.6912296

[80] M. Shahbaz, K. Shashidhar, and R. Eschbach, "Specification inference using systematic reverse-engineering methodologies: An automotive industry application," *IEEE software*, vol. 29, no. 6, pp. 62–69, 2011. doi: 10.1109/ms.2011.159. [Online]. Available: https://doi.org/10.1109/ms.2011.159

[81] A. Post, I. Menzel, J. Hoenicke, and A. Podelski, "Automotive behavioral requirements expressed in a specification pattern system: a case study at bosch," *Requirements Engineering*, vol. 17, no. 1, pp. 19–33, 2012. doi: 10.1007/s00766-011-0145-9. [Online]. Available: https://doi.org/10.1007/s00766-011-0145-9

[82] S. Konrad and B. H. Cheng, "Real-time specification patterns," in *Proceedings of the 27th international conference on Software engineering*, 2005. doi: 10.1109/icse.2005.1553580 pp. 372–381. [Online]. Available: https://doi.org/10.1109/icse.2005.1553580

[83] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st international conference on Software engineering*, 1999. doi: 10.1145/298595.298598 pp. 411–420. [Online]. Available: https://doi.org/10.1145/298595.298598

[84] H. Khosrowjerdi and K. Meinke, "Learning-based testing for autonomous systems using spatial and temporal requirements," in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, 2018. doi: 10.1145/3243127.3243129 pp. 6–15. [Online]. Available: https://doi.org/10.1145/3243127.3243129

[85] M. Kamali, S. Linker, and M. Fisher, "Modular verification of vehicle platooning with respect to decisions, space and time," in *International Workshop on Formal Techniques for Safety-Critical Systems*. Springer, 2018. doi: $10.1007/978-3-030-12988-0\_2$ pp. 18–36. [Online]. Available: https://doi.org/10.1007/978-3-030-12988-0_2

[86] M. Hilscher and M. Schwammberger, "An abstract model for proving safety of autonomous urban traffic," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2016. doi: $10.1007/978-3-319-46750-4\_16$ pp. 274–292. [Online]. Available: https://doi.org/10.1007/978-3-319-46750-4_16

[87] I. Haghighi, A. Jones, Z. Kong, E. Bartocci, R. Gros, and C. Belta, "Spatel: a novel spatial-temporal logic and its applications to networked systems," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, 2015. doi: 10.1145/2728606.2728633 pp. 189–198. [Online]. Available: https://doi.org/10.1145/2728606.2728633

[88] J.-D. Quesel and A. Schäfer, "Spatio-temporal model checking for mobile real-time systems," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2006. doi: $10.1007/11921240\_24$ pp. 347–361. [Online]. Available: https://doi.org/10.1007/11921240_24

[89] H. Khosrowjerdi, H. Nemati, and K. Meinke, "Spatio-temporal model-checking of cyber-physical systems using graph queries," in *International Conference on Tests and Proofs*. Springer, 2020. doi: $10.1007/978-3-030-50995-8\_4$ pp. 59–79. [Online]. Available: https://doi.org/10.1007/978-3-030-50995-8_4

[90] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017. doi: 10.1145/3104031. [Online]. Available: https://doi.org/10.1145/3104031

[91] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Learning-based testing for safety critical automotive applications," in *International Symposium on Model-Based Safety and Assessment*. Springer, 2017. doi: 10.1016/j.infsof.2020.106393 pp. 197–211. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106393

[92] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International conference on computer aided verification*. Springer, 2002. doi: $10.1007/3-540-45657-0\_29$ pp. 359–364. [Online]. Available: https://doi.org/10.1007/3-540-45657-0_29

[93] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009. doi: 10.1002/stvr.402. [Online]. Available: https://doi.org/10.1002/stvr.402

[94] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Automated behavioral requirements testing for automotive ecu applications," in *Proc. 5th Int. Workshop on Model Based Safety Analysis (IMBSA 2017)-to appear. Springer LNCS*, 2017. [Online]. Available: https://www.researchgate.net/profile/Karl-Meinke/publication/310599998_Automated_Behavioral_Requirements_Testing_for_Automotive_ECU_Applications/links/607d28032fb9097c0cf379e1/Automated-Behavioral-Requirements-Testing-for-Automotive-ECU-Applications.pdf

[95] A. Yasmine, A.-B. Rabea, and G.-O. Patricia, "Towards formal verification of autonomous driving supervisor functions," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02442221

[96] A. P. Chouhan and G. Banda, "Formal verification of heuristic autonomous intersection management using statistical model checking," *Sensors*, vol. 20, no. 16, p. 4506, 2020. doi: 10.3390/s20164506. [Online]. Available: https://doi.org/10.3390/s20164506

For DIVA

{
"Author1": {
"Last name": "DEI ROSSI",
"First name": "Marco",
"Local User Id": "u100001",
"E-mail": "mardr@kth.se",
"ORCiD": "0000-0001-9314-270X",
"organisation": {"L1": "School of Electrical Engineering and Computer Science ",
}
},
"Degree": {"Educational program": "Master's Programme, ICT Innovation, 120 credits"},
"Title": {
"Main title": "Automatically Learning Register Automata from MATLAB Code",
"Subtitle": "A case study in autonomous driving",
"Language": "eng" },
"Alternative title": {
"Main title": "Automatiskt Learning Register Automata från MATLAB Code",
"Subtitle": "En fallstudie i autonom körning",
"Language": "swe"
},
"Examiner1": {
"Last name": "Meinke",
"First name": "Karl",
"Local User Id": "u100004",
"E-mail": "karlm@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science ",
"L2": "Computer Science" }
},
"Cooperation": { "Partner_name": "Zenseact AB"},
"Other information": {
"Year": "2021", "Number of pages": "xviii,??"}
}

TRITA -EECS-EX-2021:743